
Preface

About this book

R has a number of packages that provide a link between the R user and a graphical toolkit, such as `tcltk`, `RGtk2` and `qtbase`. In addition, an R user can interface with Java, Python or other external languages to provide access to graphical toolkits within those languages. This book is about writing graphical user interfaces (GUI) within R that do not rely on knowing an external programming language.

The R language, like its predecessor S, is designed for interactive use through a command line interface (CLI). However, the graphical user interface (GUI) has emerged as an effective alternative, depending on the specific task and the target user base. Currently, there is a range of graphical interfaces for R that are programmed within R. For example, several package authors have provided GUIs for their functions. Examples include `limmaGUI`, `caGUI`, `clustTool`, `Metabonomic`, and others. There are a few tools to automatically generate such GUIs, such as the `fgui` package and the `guiDlgFunction` function from the `svDialogs` package. Other authors have provided graphical interfaces to explore data sets, such as `ggobi`, `playwith`, `latticeist` and `aplpack`. Still others have provided packages with GUIs aimed at allowing students to perform some simulation, e.g., `teachingDemos`. The `rattle` package provides an interface for several data mining operations. The `Rcmdr` package provides a menu- and dialog-driven interface to a wide range of R's functionality. There are several user-contributed plugins that extend the `Rcmdr`. Additionally, as R finds wider usage outside of academia, it is not uncommon for people who work in a team setting to desire an interface to their R code that allows non-R users access.

Most all of these examples are within the scope of this book. We set out to show that for many purposes adding a graphical interface to one's work is not terribly sophisticated nor time-consuming. This book does not attempt to cover the development of GUIs that require knowledge of another programming language, although several such projects exist. Many of these

are general front-ends to R, such as the Java-based GUI JGR, the rkward GUI for KDE, the biocep GUI written using Java and the RServe package, or even the Windows GUI that comes with R's Windows package. There are also several special purpose GUIs, like iPlots, which are largely implemented in Java, relying on rJava, a native interface between R and Java.

The bulk of this text covers four different packages for writing GUIs in R. The gWidgets package is covered first. This provides a common programming interface over several R packages that implement low-level, native interfaces to GUI toolkits. The gWidgets interface is much simpler – and less powerful – than the native toolkits, so is useful for a programmer who does not wish to invest too much time into perfecting a GUI. There are a few other packages that provide a high-level R interface to a toolkit such as rpanel or svDialogs, but we focus on this one.

The next three parts introduce the native interfaces upon which gWidgets is built. These offer fuller and more direct control of the underlying toolkit and thus are well suited the development of GUIs that require special features or performance characteristics. The first of these is the RGtk2 package which provides a link between R and the cross-platform GTK+ library. GTK+ is mature, feature rich and leveraged by several widely used projects.

Another mature and feature-rich toolkit is Qt, an open-source C++ library from Nokia. The R package qtbase provides a native interface from R to Qt. As Qt is implemented in C++, it is designed around the ability to create classes that extend the Qt classes. qtbase supports this from within R, although such object oriented concepts may be unfamiliar to many R users.

Finally, we discuss the tcltk package, which provides the R user access to the Tk libraries. Although not as modern as GTK+ nor Qt, these libraries come pre-installed with the Windows binary, thus avoiding installation issues for the average end-user. The bindings to Tk were the first ones to appear for R and several of the GUI projects above, notably Rcmdr, use this toolkit.

These four main parts are preceeded by an introductory chapter on GUIs and followed by a chapter on web GUIs.

The text is written with the belief that much can be learned by studying examples, and so several examples are given. Some of these are meant as sketches of what can be done, while a few illustrate how to code actual useful interfaces. This text can't expect to cover all of the features of a graphical toolkit. For the tcltk, RGtk2 and qtbase packages, the underlying toolkits have well documented APIs.

This text comes with an accompanying package ProgGUIInR. This package includes the complete code for all the examples. In order to save space, some examples in the text have code that is not shown. The package provides the functions browseGWidgetsFiles, browseRGtk2Files, browseQtFiles and browseTclTkFiles for browsing the examples from the respective chapters. Additionally, this package will contain vignettes describing aspects that did

not make it into the text.

This text was written with the Sweave package. To suppress superfluous output an assignment to a variable named QT is made at times.

Contents

Contents	iv
1 The basic ideas of Graphical User Interfaces	1
1.1 Introduction	1
1.2 A simple GUI in R	1
1.3 GUI Design Principles	5
Choice of widget	8
1.4 Controls	10
Selection	10
Checkboxes	10
Radio Button Groups	10
Sliders and spinbuttons	10
Combo boxes	11
List boxes	11
Displaying data	12
Tabular display	12
Tree widgets	12
Inititiating an action	12
Buttons	13
Icons	14
Menubars	14
Toolbars	15
Displaying and editing text	15
Single line text	15
Text edit boxes	15
Display of information	16
Labels	16
Statusbars	16
Progress bars	16
Tooltips	17
Modal dialog boxes	17

	File choosers	17
	Message dialogs	17
1.5	Containers	17
	Top level windows	18
	Box containers	19
	Frames	19
	Expanding boxes	19
	Paned boxes	20
	Grid layout	20
	Tabbed Notebooks	21
	Example	21
2	gWidgets: Overview	23
2.1	Constructors	23
2.2	Methods	25
2.3	Callbacks	26
2.4	Dialogs	28
2.5	Installation	30
3	gWidgets: Layout	33
3.1	Top-level windows	33
	A modal window	35
3.2	Box containers	36
	The ggroup container	36
	The gframe and gexpandgroup containers	39
3.3	Paned containers: the gpandedgroup container	40
3.4	Tabbed notebooks: the gnotebook container	40
3.5	Grid layout: the glayout container	42
4	gWidgets: Control Widgets	43
	Buttons	43
	Labels	44
	Statusbars	45
	Displaying icons and images stored in files	45
4.1	Text editing controls	47
	Single-line, editable text	47
	Multi-line, editable text	49
4.2	Selection controls	51
	Checkbox widget	51
	Radio button widget	52
	A group of checkboxes	52
	A combobox	53
	A slider control	54
	A spin button control	54

CONTENTS

	Selecting from the file system	55
	Selecting a date	56
4.3	Table and tree controls	56
	Display of tabular data	56
4.4	Display of heirarchical data	59
4.5	Actions, menus and toolbars	63
	Actions	63
	Toolbars	63
	Menubars, popup menus	64
5	gWidgets: R-specific widgets	67
5.1	A graphics device	67
5.2	A data frame editor	72
	Workspace browser	77
	Help browser	79
	Command line widget	79
	Simplifying creation of dialogs	79
	Laying out a form	80
	Creating a GUI for a function	84
6	RGtk2: Overview	85
6.1	Objects and Classes	86
6.2	Constructors	87
6.3	Methods	88
6.4	Properties	89
6.5	Events and signals	90
6.6	Enumerated types and flags	92
6.7	The event loop	93
7	RGtk2: Basic Components	95
7.1	Top-level windows	95
7.2	Layout containers	97
	Basics	97
	Widget size negotiation	98
	Box containers	99
	Alignment	102
7.3	Buttons	103
7.4	Static Text and Images	105
	Labels	105
	Statusbars	108
	Images	108
	Stock icons	109
7.5	Input Controls	113
	Text entry	113

Check button	114
Radio button group	114
Combo boxes	116
Sliders	118
Spin buttons	119
7.6 Containers	120
Framed containers	120
Expandable containers	120
Notebooks	120
Scrollable windows	122
Divided containers	126
Tabular layout	127
7.7 Drag and drop	129
7.8 Graphics	131
The cairoDevice package	131
8 RGtk2: Widgets Using Models	133
8.1 Display of tabular data	133
Loading a data frame	133
Displaying data as a list or table	134
Accessing GtkTreeModel	137
Selection	138
Sorting	140
Filtering	141
Cell renderer details	143
8.2 Display of hierarchical data	152
Loading hierarchical data	152
Displaying data as a tree	154
8.3 Model-based combo boxes	158
8.4 Text entry widgets with completion	161
8.5 Text views and text buffers	162
GtkTextBuffer Details	163
GtkTextView Details	169
9 RGtk2: Menus and Dialogs	171
9.1 Actions	171
9.2 Menus	172
Menubars	173
Popup Menus	174
9.3 Toolbars	175
9.4 Managing a large user interface	179
9.5 Dialogs	183
The gtkDialog constructor	183
File chooser	185

Date picker	186
10 Programming GUIs using Qt	187
10.1 An introductory example	187
10.2 The Qt library	190
10.3 Classes and objects	191
10.4 Methods and dispatch	193
10.5 Properties	194
10.6 Signals	195
10.7 Enumerations and flags	196
10.8 Defining Classes and Methods	197
10.9 Common methods for QWidget and QObject	201
Fonts	203
Styles	203
Style Sheets	204
10.10 Drag and drop	204
11 Layout managers	209
11.1 Box layouts	212
Scrolling layouts	214
Framed Layouts	215
Separators	215
11.2 Grid Layouts	215
11.3 Form layouts	217
11.4 Notebooks	218
QStackedWidget	219
11.5 Paned Windows	220
11.6 Main windows	220
Actions	220
Menubars	222
Context menus	223
Toolbars	225
Statusbars	226
Dockable widgets	226
12 Widgets	229
12.1 Dialogs	229
Message Dialogs	229
Input dialogs	231
QDialog	232
File and Directory choosing dialogs	233
12.2 Labels	234
12.3 Buttons	235
Button boxes	235

Icons and pixmaps	236
12.4 Checkboxes	237
Groups of checkboxes	237
12.5 Radio groups	238
12.6 Sliders and spin boxes	239
Sliders	239
Spin boxes	240
12.7 Single-line text	241
Completion	242
Masks and Validation	242
12.8 Multi-line text	243
13 Widgets using the MVC framework	251
13.1 Model View Controller implementation in Qt	251
13.2 The item-view classes	252
Comboboxes	252
A list widget	254
A table widget	258
A Tree Widget	265
13.3 Item models and their views	271
Using a data frame for a model	271
Table Models	272
Table views	273
Custom Views	278
14 Qt paint	281
15 Tcl Tk: Overview	283
15.1 Interacting with Tcl	284
15.2 Constructors	286
Geometry managers	288
Tcl variables	289
Colors and fonts	289
Images	291
Themes	292
Window properties and state: tkwininfo	293
15.3 Events and Callbacks	294
Callbacks	294
Events	295
% Substitutions	296
16 Tcl Tk: Containers and Layout	301
16.1 Top-level windows	301
16.2 Frames	303

CONTENTS

Label Frames	303
16.3 Geometry Managers	304
Pack	304
Grid	309
16.4 Other containers	314
Paned Windows	315
Notebooks	316
17 Tcl Tk: Widgets	319
17.1 Selection Widgets	319
Checkbutton	319
Radio Buttons	320
Comboboxes	321
Scale widgets	322
Spinboxes	323
17.2 Text widgets	327
Entry Widgets	327
Scrollbars	330
Multi-line Text Widgets	331
17.3 Treeview widget	335
Rectangular data	335
Heirarchical data	341
17.4 Menus	344
17.5 Canvas Widget	348
17.6 Dialogs	354
Modal dialogs	354
File and directory selection	354
Choosing a color	355
Bibliography	357

The basic ideas of Graphical User Interfaces

1.1 Introduction

There are many reasons why one would want to create a graphical user interface for a package or piece of R functionality. For example,

- GUIs can make R's functionality available to the casual R user,
- GUIs can be dynamic, they can direct the user how to fill in arguments, can give feedback on the choice of an argument, they can prevent or allow user input as appropriate,
- Although a command line is usually faster when the commands are known, a GUI can make some less commonly used tasks easier to do.
- GUIs make dealing with large data sets easier both visually and as an alternative to sometimes difficult command line usage.
- GUIs can tightly integrate with graphics, for example the `rggobi` interface among others.

Even as R is rapidly gaining interest, especially commercial interest, it lacks a common graphical user interface. This is due to several reasons. (1) The R language is designed for command line usage. (2) The GUI would need to run on all supported R platforms (3) the wide variety of user types means a single interface would be unlikely to satisfy all. Even if it did have a common interface, as much of R's functionality depends on user contributions there will always be a demand for package programmers to provide convenient interfaces to their work.

1.2 A simple GUI in R

We begin with an example showing how one can use R's standard graphics device, as a canvas for drawing a GUI, for playing a game of tic-tac-toe against the computer. Although this example has nothing to do with statistics, it illustrates, in a familiar way, some of the issues that this text will address with using GUIs.

1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

Many GUIs can be thought of as different views of some data model. In this example, the data simply consists of information holding the state of the game. Here we define a global variable, `board`, to store the current state of the game.

```
board <- matrix(rep(0,9), nrow=3)      # a global
```

The GUI provides the representation of the data for the user. This example just uses a canvas for this, but most GUIs have a combination of components to represent the data and allow for user interaction. The layout of the GUI directs the user as to how to interact with it and is an important factor as to whether the GUI will be well received. Here we define a function to layout the game board using the graphics device as a canvas.

```
layoutBoard <- function() {  
  plot.new()  
  plot.window(xlim=c(1,4), ylim=c(1,4))  
  abline(v=2:3); abline(h=2:3)  
  mtext("Tic Tac Toe. Click a square:")  
}
```

A GUI is designed to respond to user input typically by the mouse or keyboard. The underlying toolkit allows the programmer to assign functions to be called when some specific event occurs. Typically, the toolkit *signals* that some action has occurred, and then calls *callbacks* or *event handlers* that have been assigned by the programmer. How this is implemented varies from toolkit to toolkit.

R's interactive graphics devices implement the `locator` function which responds to mouse clicks by user. When using this function, one specifies how many mouse clicks to gather and the *control* of the program is suspended until these are gathered (or the process is terminated). The suspension of control makes this a *modal* GUI. This design is common for simple dialogs that require immediate user attention, but not common otherwise. To make non-modal dialogs possible, the writers of the R packages that interface with the GUI toolkits have to interface with R's event loop mechanism.

Here we define a function to collect the player's input.

```
doPlay <- function() {  
  iloc <- locator(n=1, type="n")  
  clickHandler(iloc)  
}
```

In the above function, `clickHandler` is an *event handler*. Its job is to process the output of the `locator` function, checking first if the user terminated `locator` using the keyboard. If not it proceeds to draw the move, and then, if necessary, the computer's move. Afterwards, play is repeated until there is a winner or a "cat's" game.

```
clickHandler <- function(iloc) {
```

```

if(is.null(iloc))
  stop("Game terminated early")
move <- floor(unlist(iloc))
drawMove(move,"x")
board[3*(move[2]-1) + move[1]] <<- 1
if(!isFinished())
  doComputerMove()
if(!isFinished())
  doPlay()
}

```

The use of `<<-` in the handler illustrates a typical issue in GUI design within R. After a GUI is created, the state is typically modified within the scope of the callback functions. These callbacks need to be able to modify values outside of their scope, yet even if the values are passed in as argument, this is usually not possible while assigning within the scope of the function call, due to R's pass by copy function calls. As such, global variables are often employed along with some strategies to avoid an explosion of variable names.

Validation of user input is an important task for a GUI, especially for Web GUIs. In the above, the `clickHandler` function checks to see if the user terminated the game early and issues a message, more helpful forms of validation are possible in general.

At this point, we have a data model, a view of the model and the logic that connects the two, but we still need to implement some of the functions to tie it together.

This function draws either an "x" or an "o". It does so using the `lines` function. The `z` argument contains the coordinates of the square to draw.

```

drawMove <- function(z,type="x") {
  i <- max(1,min(3,z[1])); j <- max(1,min(3,z[2]))
  if(type == "x") {
    lines(i + c(.1,.9),j + c(.1,.9))
    lines(i + c(.1,.9),j + c(.9,.1))
  } else {
    theta <- seq(0,2*pi,length=100)
    lines(i + 1/2 + .4*cos(theta), j + 1/2 + .4*sin(theta))
  }
}

```

One could use `text` to place a text "x" or "o", but this may not look good if the GUI is resized. Most GUI layouts allow for dynamic resizing. Overall this is a great advantage, for example, it allows translations to just worry about the text and not the layout, which will be different for every language.

This function is used to test if a game is finished. The matrix `m` allows us to easily check all the possible ways to get three in a row.

```

isFinished <- function() {

```

```
if(sum(abs(board)) >= 9)
  return(TRUE)
m <- matrix(1:9,nrow=3)
ways <- list(m[,1], m[,2], m[,3],
             m[1,], m[2,], m[3,],
             diag(m),diag(apply(m,2,rev)))
sums <- sapply(ways, function(i) abs(sum(board[i])))
if(any(sums == 3))
  return(TRUE)
return(FALSE)
}
```

This function picks a move for the computer. The move is converted into coordinates using %% to get the remainder and %/% to get the quotient when dividing an integer by an integer. This function just chooses at random from the left over positions; we leave implementing a better strategy to the interested reader.

```
doComputerMove <- function() {
  newMove <- sample(which(board == 0),1) # random !
  board[newMove] <- -1
  z <- c((newMove-1) %% 3, (newMove-1) %/% 3) + 1
  drawMove(z,"o")
}
```

This function provides the main entry point for our GUI. To play a game it first lays out the board and then calls doPlay. When this function terminates, a message is written on the screen.

```
playGame <- function() {
  layoutBoard()
  doPlay()
  mtext("All done\n",1)
}
```

This example adheres to the model-view-controller design pattern that is implemented by virtually every complex GUI. We will encounter this pattern throughout this book, although it is not always explicit.

A final point, the above example illustrates a common issue when designing software that is particularly true of GUIs – feature creep is an endless temptation. In this case, there are many obvious improvements: localizing the text messages so different languages can be used, implementing a better logic for the computer’s moves, drawing a line connecting three in a row when there is a win, indicating who won when there is a win, etc. For many GUIs there is a necessary trade-off between usability and complexity.

1.3 GUI Design Principles

The most prevalent pattern of user interface design is denoted WIMP, which stands for Window, Icon, Menu and Pointer (i.e., mouse). The WIMP approach was developed at Xerox PARC in the 1970's and later popularized by the Apple Macintosh in 1984. This is particularly evident in the separation of the window from the menubar on the Mac desktop. Other graphical operating systems, such as Microsoft Windows, later adapted the WIMP paradigm, and libraries of reusable GUI components emerged to support development of applications in such environments. Thus, GUI development in R adheres to WIMP approach.

The primary WIMP component from our perspective is the window. A typical interface design consists of a top-level window referred to as the *document window* that shows the current state of a “document”, whatever that is taken to be. In R it could be a data frame, a command line, a function editor, a graphic or an arbitrarily complex form containing an assortment of such elements.

Abstractly, WIMP is a command language, where the user executes commands, often called actions, on a document by interacting with graphical controls. Every control in a window belongs to some abstract menu. Two common ways of organizing controls into menus are the menubar and toolbar.

The parameters of an action call, if any, are controlled in sub-windows. These sub-windows are termed *application windows* by Apple (?), but we prefer the term *dialogs*, or *dialog boxes*. These terms may also refer to smaller sub-windows that are used for alerts or confirmation. The program often needs to wait for user input before continuing with an action, in which case the window is modal. We refer to these as *modal dialog boxes*.

Each window or dialog typically consists of numerous controls laid out in some manner to facilitate the user interaction. Each window and control is a type of *widget*, the basic element of a GUI. Every GUI is constituted by its widgets. Not all widgets are directly visible by the user; for example, many GUI frameworks employ invisible widgets to lay out the other widgets in a window.

There is a wide variety of available widget types, and widgets may be combined in an infinite number of ways. Thus, there are often numerous means to achieve the same goals. For example, Figures 1.1 and 1.2 show three dialogs that perform the same task – collect arguments from the user to customize the printing of a document. Although all were designed to do the same thing, there are many differences in implementation.

In some cases, typical usage suggests one control over another. The choice of printer for each is specified through a combo box. However, for other choices a variety of widgets are employed. For example, the control to indicate the number of copies for the Mac is a simple text entry window, whereas

1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

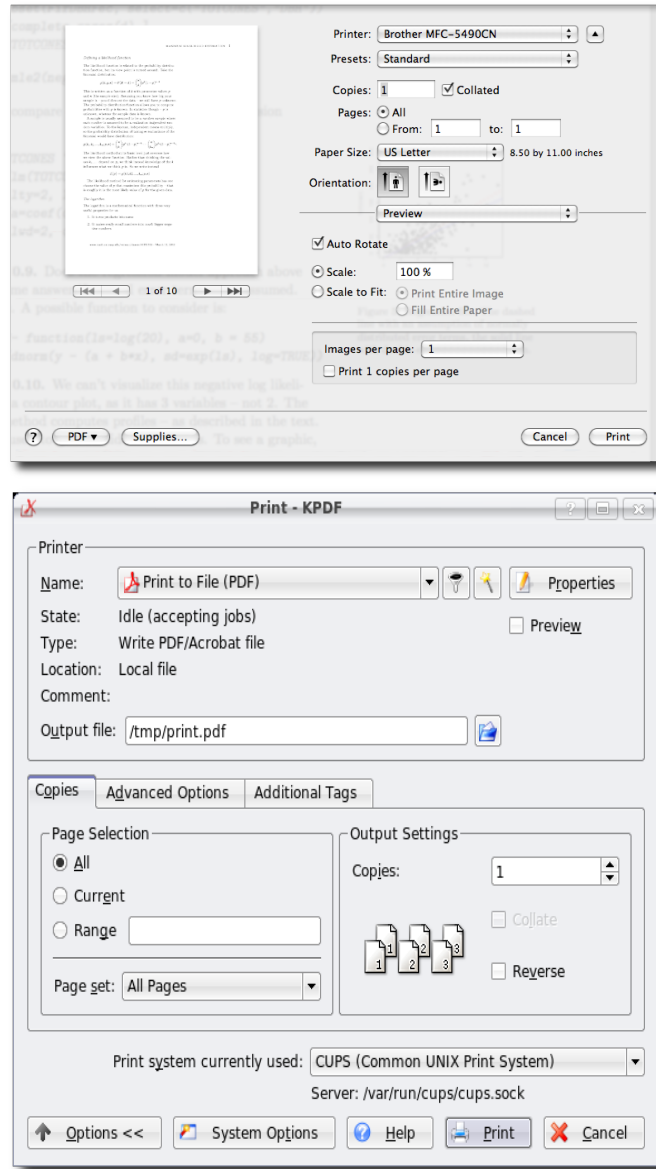


Figure 1.1: Two print dialogs. One from Mac OS X 10.6 and one from KDE 3.5.

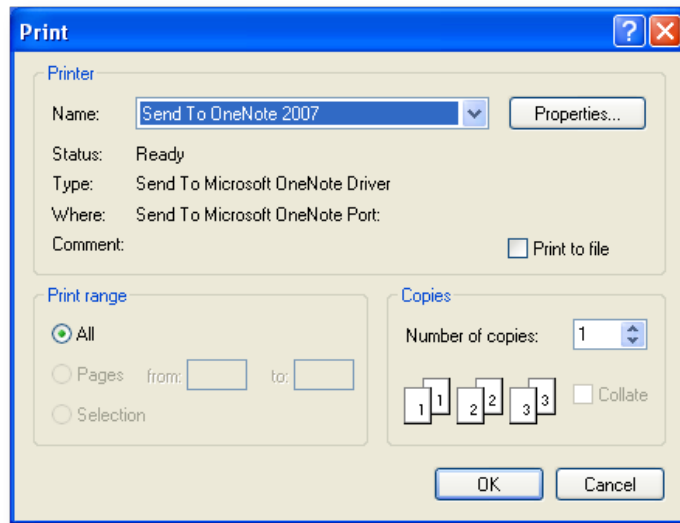


Figure 1.2: R's print dialog under windows XP using XP's native dialog.

for the KDE and R dialog it is a spinbutton. The latter minimizes user error, say through entering a non-positive integer. The KDE and Mac dialogs have icons to compactly represent actions, whereas the R example has none. The landscape icon for the Mac is very clear and provides this feature without having to use a sub dialog.

How the interfaces are laid out also varies. All panels are read top to bottom, although the Mac interface also has a very nice preview feature on the left side. The KDE dialog uses frames to separate out the printer arguments from the arguments that specify how the print job is to proceed. The Mac uses a vertical arrangement to guide the user through this. For the Mac, horizontal separators are used instead of frames to break up the areas, although a frame is used towards the bottom. Apple uses a center balance for its controls. They are not left justified as are the KDE and Windows dialogs. Apple has strict user-interface guidelines and this center balance is a design decision.

The layout also determines how many features and choices are visible to the user at a given time. For example, the Mac GUI uses “disclosure buttons” to allow access to printer properties and the PDF settings, whereas KDE uses a notebook container to show only a subset of the options at once.

The Mac GUI provides a very nice preview of the current document indicating to the user clearly what is to be printed and how much. Adjusting GUIs to the possible state is an important user interface property. GUI areas that are not currently sensitive to user input are grayed out. For example, the

“collate” feature of the GUI only makes sense when multiple copies are selected, so the designers have it grayed out until then. A common element of GUI design is to only enable controls when their associated action is possible, given the state of the application.

The Mac GUI has the number of pages in focus, whereas Windows places the printer in focus. This allows the user to interact with the GUI without the mouse. Typically the tab key is used to step through the controls. GUI's often have keyboard accelerators that allow power users to shift the focus directly to a specific widget. Examples are found in the KDE dialog, where the underlined letters indicate the accelerator key. Most dialogs also have a default button, which will initiate the dialog action when the return key is pressed. The KDE dialog, for example, indicates that the “print” button is the default button through special shading.

Each dialog presents the user with a range of buttons to initiate or cancel the printing. The Windows ones are set on the right and consist of the standard “OK” and “Cancel” buttons. The Mac interface uses a spring to push some buttons to the left, and some to the right to keep separate their level of importance. The KDE buttons do so as well, although one can't tell from this. However, one can see the use of stock icons on the buttons to guide the user.

Choice of widget

A GUI is comprised of one or more widgets or controls. The appropriate choice of widget depends on a balance of considerations. For example, many widgets offer the user a selection from one or more possible choices. An appropriate choice depends on the type and size of the information being displayed, the constraints on the user input, and on the space available in the GUI layout. As an example, Table 1.3 lists suggests different types of widgets used for this purpose depending on the type and size of data and the number of items to select.

Figure 1.3 shows several such controls in a single GUI. A checkbox enables an intercept, a radio group selects either full factorial or a custom model, a combobox selects the “sum of squares” type, and a list box allows for multiple selection from the available variables in the data set.

For many R object types there are natural choices of widget. For example, values from a sequence map naturally to a slider or spin button; a data frame maps naturally to a table widget; or a list with similar structure can map naturally to a tree widget. However, certain R types have less common metaphors. For instance, a formula object can be fairly complex. Figure 1.3 shows an SPSS dialog to build terms in a model. R power users may be much faster specifying the formula through a text entry box, but beginning R users coming to grips with the command line and the concept of a formula may benefit from the assistance of a well designed GUI. One might desire an

Table 1.1: Table of possible selection widgets by data type and size

Type of data	Single	Multiple
Boolean	Checkbox	
Small list	radiogroup combobox list box	checkboxgroup list box
Moderate list	combobox list box	list box
Large list	list box	list box
Sequential	slider spinbutton	
Tabular	table	table
Heirarchical	tree	tree

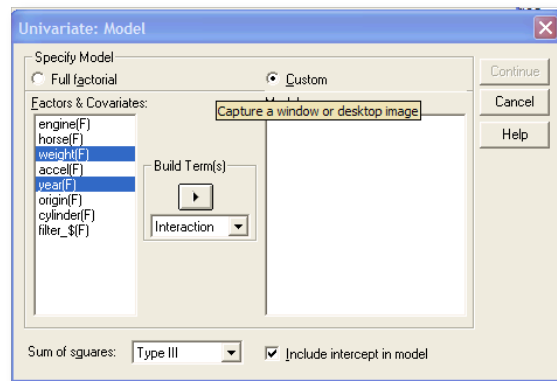


Figure 1.3: A dialog box from SPSS version 11 for specifying terms for a linear model. The graphic shows a dialog that allows the user to specify individual terms in the model using several types of widgets for selection of values, such as a radio group, a checkbox, combo boxes, and list boxes.

interface that balances the needs of both types of user, or the SPSS interface may be appropriate. Knowing the potential user base is important.

1.4 Controls

This section provides an overview of many common controls, i.e., widgets that represent the actions to be performed on a document. Each displays some information, and most accept user input.

Selection

A common task for a GUI is the selection of a value. In the context of R, there are many different types of values the user may need to select. For example, selecting a data frame from a list of data frames, selecting a variable in a data frame, selecting certain cases in a data frame, selecting a logical value for a function argument, selecting a numeric value for a confidence level or selecting a string to specify an alternative hypothesis. Clearly there can be no one-size-fits-all widget to handle the selection of a value. We describe some standard selection widgets next.

Checkboxes

A *checkbox* allows the user to select a logical value for a variable. Checkboxes have labels to indicate which variable is being selected. Combining multiple checkboxes into a group allows for the selection of one or more values at a time.

Radio Button Groups

A *radio button group* allows a user to select exactly one value from a vector of possible values. The analogy dates back to old car radios where there were a handful of buttons to press to select a preset channel. When a new button was pushed in, the old button popped up. This safety feature allowed drivers to keep their eyes on the road. Radio button groups are useful, provided there are not too many values to choose from, as all the values are shown. These values can be arranged in a row, a column or both rows and columns to better use screen space.

Sliders and spinbuttons

A *slider* is a widget that selects a value from a sequence of possible values (typically) through the manipulation of a knob that can visually range over the possible values. Some toolkits (e.g. Java/Swing) only allow for the sequence to have integer values. The slider is a good choice for offering the user a selection of parameter values. The `tkdensity` demo of the `tk` package

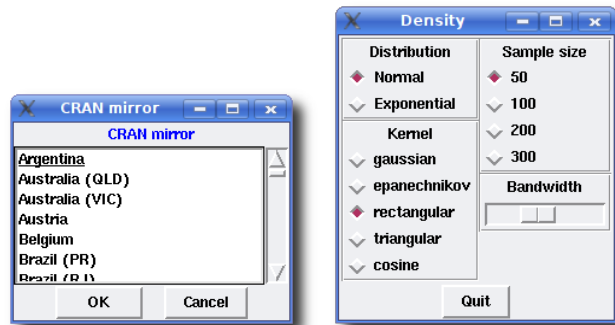


Figure 1.4: Two applications of the tcltk package. The left graphic is produced by `chooseCRANmirror` and uses a list box to allow selection from a long list of possibilities. The right graphic is the `tkdensity` demo from the package. It uses radio buttons and a slider to select the parameter values for a density plot.

(Figure 1.4) uses a slider to dynamically adjust the bandwidth of a density estimate.

A *spin button* also allows the user to specify a value from a possible sequence of values. Typically, this widget is drawn with a text box displaying the current value and two arrows to increment or decrement the selection. The text box can usually be edited. Some toolkits generalize beyond a numeric sequence. For example, the letters of the alphabet could be a sequence. A spin button has the advantage of using less screen space, but is less usable if the sequence is long, although often the user can enter in the choice using the keyboard. A spin button is used in the KDE print dialog of Figure 1.1 to adjust the number of copies.

Combo boxes

A *combo box* is a widget that allows the selection of one of several fixed values, while displaying just the currently selected one. Comboboxes may also offer the user the ability to specify a value, in which case they are combined with a text entry area. From a screen-space perspective, they can efficiently allow the selection of a value from many values, although a choice from too many values can be annoying to the user, such as when a web form requests the selection of a country from hundreds of choices.

List boxes

A *list box* is a widget that displays in a column the list of possible choices. A scrollbar is used when the list is too long to show in the allocated space.

Some toolkits have list boxes that allow the values to spread out over several columns. Selection typically occurs with a right mouse click or through the keyboard, whereas a double-click will typically be programmed to initiate some action. Unlike comboboxes, list boxes can be used for multiple selection. This is typically done by holding down either the shift or ctrl keys. Figure 1.4 shows a list box created by R that is called from the function `chooseCRANmirror`.

Displaying data

Table and tree widgets support the display and manipulation of tabular and hierarchical data, respectively. More arbitrary data visualization, such as statistical plots, can be drawn within a GUI window, but such is beyond the scope of this section.

Tabular display

A *table widget* shows tabular data, such as a data frame, where each column has a specific data type and cell rendering strategy. Table widgets handle the display, sorting and selection of records from a dataset, and may support editing. Figure 1.5 shows a table widget being used in a Spotfire web player demonstration to display the cases that a user selects through the filtering controls.

Tree widgets

So far, we have seen how list boxes display homogeneous vectors of data, and how table widgets display tabular data, like that in a data frame. Other widgets support the display of more complex data structures. If the data has a hierarchical structure, then a *tree widget* may be appropriate for its display. Examples of hierarchical data in R, are directory structures, the components of a list, or class hierarchies. The object browser in JGR uses a tree widget to show the components of the objects in a users session (the left graphic of Figure 1.6). The root node of the tree is the “data” folder, and each data object in the global workspace is treated as an offspring of this root node. For the data frame `iraq`, its variables are considered as offspring of the data frame. In this case these variables have no further offspring, as indicated by the “page” icon.

Initiating an action

After the user has specified the parameters of an action, typically through the selection widgets presented above, it comes time to execute the action. Widgets that execute actions include the familiar buttons, menubars and toolbars.

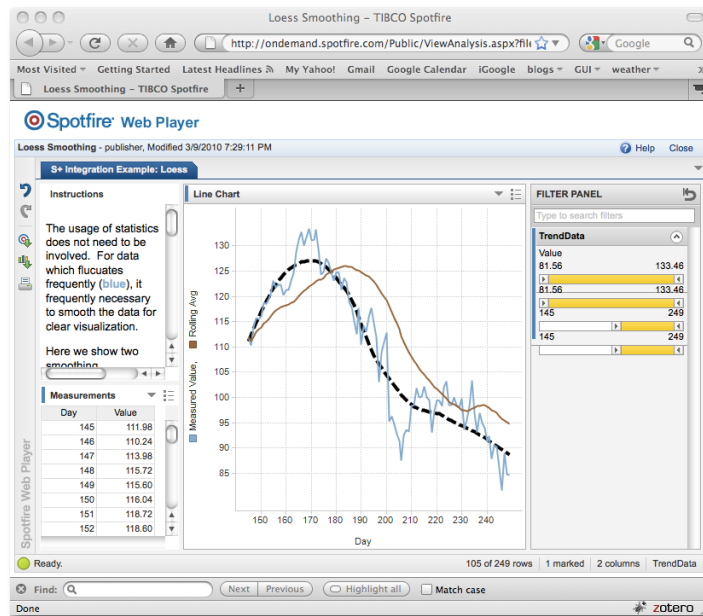


Figure 1.5: A screen shot from Tibco's Spotfire web player illustrating a table widget (lower left) being used to display the selected cases that are summarized in the graphic. The right bar provides a means to filter the cases under consideration.

Buttons

A *button* is typically used to give the user a place to click the mouse in order to initiate some immediate action. The "Properties" button, when clicked, opens a dialog for setting printer properties. The button with the wizard icon also opens a dialog. As buttons typically lead to an action, they often are labeled with a verb. (?) In Figure 1.3 we see how SPSS uses buttons in its dialogs: buttons which are not valid in the current state are disabled; buttons which are designed to open subsequent dialogs have trailing dots; and the standard actions of resetting the data, canceling the dialog or requesting help are given their own buttons on the right edge of the dialog box.

To speed the user through a dialog, a button may be singled out as the default button, so its action will be called if the user presses the return key. As well, buttons may be given accelerator key bindings, so as their actions are accessible by typing the proper key combination. The KDE print dialog in Figure 1.1 has these bindings indicated through the underlined letter on the button label's.

Icons

In the WIMP paradigm, an *icon* is a pictorial representation of a resource, such as a document or program, or, more generally, a concept, such as a type of file. An application GUI typically adopts the more general definition, where an icon is used to augment or replace a text label on a button, a toolbar, in a list box, etc. When icons are used on toolbars and buttons, they are associated with actions, so the icons should have some visual implication of an action. Well chosen icons make a big visual difference in a GUI.

Menubars

Xerox Parc's revolutionary idea of a WIMP GUI added windows, icons, menubars, and pointing devices to the desktop computing environment. The central role of menu bars has not diminished. For a GUI, the *menubar* gives access to the full range of functionality available. Each common action should have a corresponding menu item. Menubars are traditionally associated with a top-level window. This is enforced by the toolkit in wxWidgets and Java but not Tcl/Tk and GTK+. In Mac OS X, the menubar appears on the top line of the display, but otherwise they typically appear at the top of the main window. In most modern applications, standard document-based design is used to organize a GUI and its actions, with a main window showing the document and its menu bar calling actions on the document, some of which may need to open subsequent application windows or dialogs for gathering additional user input. In this model, only the main window has a menu bar, not the application windows or dialogs. In a statistics application, the "document" may be viewed as the active data frame, a report, or a graphic.

The styles used for menubars are fairly standardized, as this allows new users to quickly orient themselves with a GUI. The visible menu names are often in the order File, Edit, View, Tools, then application specific menus, and finally a Help menu. Each visible menu item when clicked opens a menu of possible actions. The text for these actions traditionally use a ... to indicate that a subsequent dialog will open so that more information can be gathered to complete the action, as opposed to some immediate action being taken. The text may also indicate a key-board accelerator, such as Find Next F3 indicating that both "N" as a keyboard accelerator and F3 as a shortcut will initiate this same action.

Not all actions will be applicable at any given time. It is recommended that rather than deleting these menu items, they be disabled, or greyed out, instead.

Menus can get very long. To help the user navigate, menu items are usually grouped together, first by being under the appropriate menu title, then with either horizontal separators to define subsequent groupings, or hierarchical submenus. The latter are indicated with an arrow. Several different

levels are possible, but navigating through too many can be tedious.

The use of menus has evolved to also allow the user to set properties or attributes of current state of the GUI. There may be checkboxes drawn next to the menu item or some icon indicating the current state.

Another use of menus is to bind contextual menus (popup menus) to certain mouse clicks on GUI elements. Typically right mouse clicks will pop up a menu that lists often-used commands that are appropriate for that widget and the current state of the GUI. In Mac OS X one-button users, these menus are bound to a control-click.

Toolbars

Toolbars are used to give immediate access to the frequently used actions defined in the menubar. Toolbars typically have icons representing the action and perhaps accompanying text. They traditionally appear on the top of a window, but sometimes are used along the edges.

Displaying and editing text

As much as possible, WIMP GUIs are designed around using the pointing device to select values for user input. Perhaps this is because it is difficult to both type and move the mouse at the same time. For statistical GUIs, especially for R with its powerful command line, the flexibility afforded by arbitrary text entry is essential for any moderately complex GUI. There is a distinction made between widgets for handling just single lines versus multiple lines of text.

Single line text

A text entry widget for editing a single line of text is used in the KDE print dialog (Figure 1.1) to specify the page range. As range's can be complex to specify, the command line has an advantage. A disadvantage of using this type of widget is the need to validate the user's input, as the input must conform to some specification.

Text edit boxes

Multi-line text entry areas are used in many GUIs. The right graphic of Figure 1.6 shows a text entry area used by Rcmdr to enter R commands, to show the output of the commands and to provide a message area (in lieu of a status bar). The "Output Window" demonstrates the utility of formatting attributes. In this case, attributes are used to specify the color of the commands, so that the input can be distinguished from the output.

1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

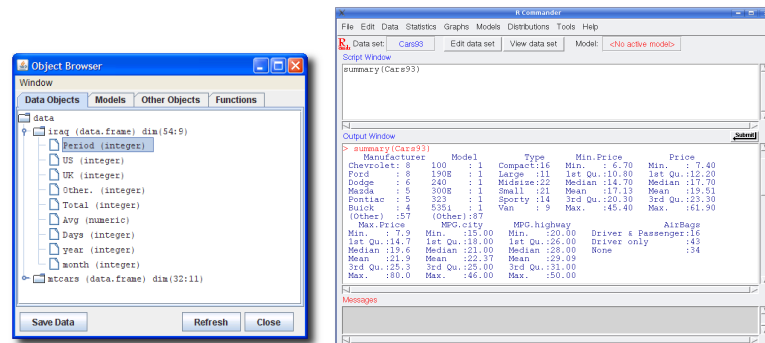


Figure 1.6: Some windows from R GUIs. The left graphic shows the object browser in the JGR GUI using a tree widget to display the possibly hierarchical nature of R objects. The right graphic shows the main Rcmdr (1.3-11) window illustrating the use of multi-line text entry areas for a command area, an output area and a message area.

Display of information

Some widgets are typically used to just display information and often do not respond to mouse clicks. These are called static controls in wxWidgets.

Labels

A label is a widget for placing text into a GUI that is typically not intended for editing, or even selecting with a mouse. This widget is used to label other controls, so the user understands what will happen when that control is changed.

A Label's text can be marked up in some toolkits. Figure ?? shows labels marked in red and blue in `tc1tk`.

Statusbars

A typical top-level window will have a menubar and toolbar for access to the possible actions, an area to display the document being worked on, and at the bottom of the window a statusbar for giving the user immediate feedback on the actions that have been initiated.

Progress bars

A progress bar is used to indicate the percentage of a particular task that has been completed. They are often used during software installation.

Tooltips

A tooltip is a small window that pops up when a user hovers their mouse over a tooltip-enabled widget. These are useful for providing extra information about a particular piece of content displayed by a widget. A common use-case is to guide new users of a GUI. Many toolkits support the display of interactive hypertext in a tooltip, which allows the user to request additional details.

Modal dialog boxes

A *modal dialog box* is a dialog box that keeps the focus until the user takes an action to dismiss the box. They are used to notify the user of some action, perhaps asking for confirmation in case the action is destructive, such as overwriting of a file name. Modal dialog boxes can be disruptive to the flow of interaction, so they should be used sparingly. As the flow essentially stops until the window is dismissed, functions that call modal dialogs can return a value when an event occurs, rather than have a handler respond to the dismiss event. The `file.choose` function, mentioned below, is a good example. When used during an interactive R session, the user is unable to interact with the command line until a file has been specified and the dialog dismissed.

File choosers

A file chooser allows for the selection of existing files, existing directories, or specifying the name of a new file. They are familiar to any user of a GUI. A typical R installation has the functions `file.choose` and `tkchooseDirectory` (in the `tcltk` package) to select files and directories.

Other common choosers are color choosers and font choosers.

Message dialogs

A *message dialog* is a high-level dialog widget for communicating a message to the user. Generally, it has a standard form. There is a small rectangular box that appears in the middle of the screen with an icon on the left and a message on the right. At the bottom is a button to dismiss the dialog, often labeled “OK.” The *confirmation dialog* variant would add a “Cancel” button which invalidates the proposed action.

1.5 Containers

Widgets in a GUI are organized in a *widget heirarchy*, where some widgets are parents and some children (which may in turn be parents). The top-level

window may play a special role here, as a parent but not a child. This organization offers the toolkit writers the chance to treat each object as a standalone component. For example, when a GUI is resized the typical algorithm is for the parent to be resized, then to send a signal to the children to resize themselves. Another example is when a window is closed, prior to closing the window will signal its children that they will be destroyed, and they should in turn signal their children, if any. A means to traverse the widget hierarchy is provided by each toolkit. In `tcltk` this hierarchical relationship is explicit, as a widget constructor – except for top-level windows – requires a parent widget when be constructed. In the other toolkits, it may be implicit.

In the widget hierarchy, the parents play a different role as those widgets that are just children. The parents play the role of *containers*. Sometimes the word *widget* is reserved for GUI components which are not-containers. Having containers makes it possible to organize GUIs into individual components – again, a desirable design feature.

The children of the GUI must be organized in some manner. In `GTK+(gWidgets)`, this is done through the choice of container with parameters being set to adjust the placement within the container when the child is added. In `Qt` and `Tcl/Tk` there is an added abstraction of a layout. A layout decouples the hierarchy from the layout and offers much more flexibility. In `Qt` this allows for a richer set of default layout options, and the ability to create ones specialized layouts.

Top level windows

The top-level window of a GUI is typically decorated with a title and buttons to iconify, maximize, or close. Besides the text of the title, the decorations are generally the domain of the window manager, often part of the operating system. The application controls the contents of the window. Generally, a top-level window will consist of a menu bar, a tool bar and a status bar. In between these is the area referred to as the *client area* or *content pane* where other containers or widgets are placed.

The title is a property of the window and may be specified at the time of construction or afterwards.

On a desktop, only one window may have the focus at a time. It may or may not be desired that a new window receive the focus so some means to specify the focus at construction or later is provided by the toolkit.

The initial placement of the window also may be specified at the time of construction. The top-level window of a GUI may generally be placed wherever it is convenient for the user, but sub-windows are often drawn on top of their parent window, as are modal dialog boxes.

The window manager usually decorates a top-level window with a close button. It may be necessary or desirable to specify some action to take place when this button is clicked. For instance, a user might be prompted if they

wish to save changes to their work when the close button is pressed.

it may take some time to initially layout a top-level window. Rather than have the window drawn and then have a blank window while this time passes, it is preferable to not make the window visible until the window is ready to draw.

We now describe some of the main containers.

Box containers

A box container places its children in it from left to right or from top to bottom. If each child is viewed as a box, then this container holds them by packing them next to each other. The upper left figure in Figure 1.7 illustrates this.

When the boxes have different sizes, then some means to align them must be decided on. Several possibilities exists. The alignment could be around some center, aligned at a baseline, the top line, or each child can specify where to anchor itself within the allotted space (the upper right graphic in Figure 1.7).

If the space allocated for a box is larger than the space requested by a child component then a decision as to how that component gets placed needs to be made. If the component is not enlarged, then there are nine reasonable places – the center and the 8 compass directions N, NW, W, Otherwise, it may be desirable for the component to expand horizontally, vertically or both (the lower left graphic in Figure 1.7). Additionally, it is desirable to be able to place a fixed amount of space between child components or a spring between components. A spring forces all subsequent to children to the far right or bottom of the container (the lower right graphic in Figure 1.7).

When a top-level window is resized, these space allocations must be made. To help, the different toolkits allows the components to have a size specified. Some combination of a minimum size, a preferred size, a default size, a specific size, or a maximum size are allowed. Specifying fixed sizes for components is generally frowned upon, as they don't scale well when a user resizes a window and they don't work well when different languages are used on the controls when an application is localized.

Frames

A box container may have a border drawn around it to visually separate its contents from others. This border may also have a title. In GTK+ these are called frames, but this word is reserved in Tcl/Tk and Java.

Expanding boxes

In order to save screen space, a means to hide a boxes contents can be used. This hiding/showing is initiated by a mouse click on a disclosure button or

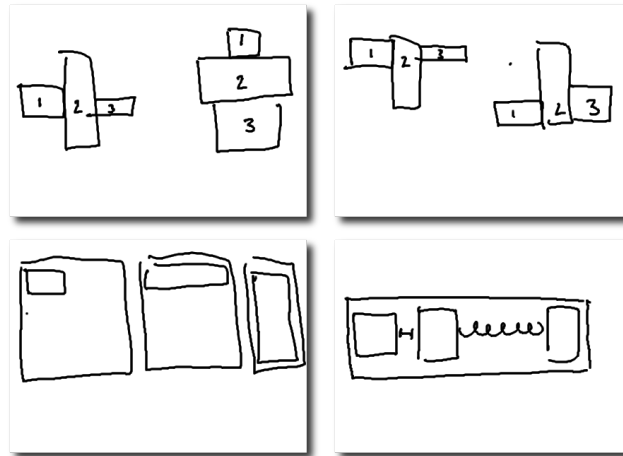


Figure 1.7: XXX REPLACE WITH REAL GRAPHICS XXX. Different possibilities for packing child components within a box. The upper left shows horizontal and vertical layout. The upper right shows some possible alignments. The lower left shows that a child could possibly be anchored in one of 9 positions. As well, it could “expand” to fill the space either horizontally (as shown) or vertically, or both. The lower right shows both a fixed amount of space between the children and an expanding spring between the child components.

trigger icon.

Paned boxes

If automatic space allocation between two child components is not desired, but rather a means for the user to allocate the space is, then a *paned container* may be used. These offer one or more horizontal or vertical sash that can be clicked and dragged to apportion space between the child components.

Grid layout

By nesting box containers, a great deal of flexibility in layout can be achieved. However, there is still a need for the alignment of child components in a tabular manner. The most flexible alignments allow for different sizes for each column and each row, and additionally, the ability for the child components to span multiple columns or rows. Within each cell (or cells) the placement of a child component mirrors that of the lower left graphic of Figure 1.7. Some specification where to anchor the component when there are nine possible positions plus expanding options must be made.

Tabbed Notebooks

A notebook is a common container to hold one or more pages (or children). The different pages are shown by the user through the clicking of a corresponding tab. The metaphor being a tabbed notebook. Modern web browsers take advantage of this container to allow several web pages to be open at once.

Example

The KDE print dialog of Figure 1.1 shows most of the containers previously described.

The top-level window has the generic title “Print – KPDF.” This window appears to have four child components: a frame labeled `Printer`, a notebook with open tab `Copies`, a grid layout for specifying the print system, and a box for holding five buttons at the bottom.

The lower left `Options` button has << to indicate that clicking this will close an expanding box, in this case a box that contains the lower three components above. So in fact, there are two visible child components of the top-level window.

The framed box holds a grid layout with five columns and 6 rows. The sizes allocated to each column are visible in the first row. It is quickly seen that each column has a different size. The last row has a text entry area that spans the second and third columns. The first column has only labels. These are anchored to the left side of the allowed space. The Apple human interface guide (?) suggests using colons for text that provides context for controls, and the KDE designers do to.

The displayed page of the notebook shows two child components, both framed boxes. A pleasant amount of space between the frames and their child components has been chosen. The `Page Selection` frame has components including radio buttons, a text area, a horizontal separator, and a combo box.

The print system information is displayed in a grid layout that has been right aligned within its parent container – the expanded group, but its children are center-balanced with the label “Print system currently used” is right aligned and “Server...” is left aligned within their cells.

The button box shows five buttons as child components. At first glance the sizing appears to show that each button is drawn to fully show its label with some fixed space placed between the buttons. If the dialog is expanded, it is seen that there is a spring between the 3rd and 4th buttons, so that the first 3 are aligned with the left side of the window and the last two the right side.

gWidgets: Overview

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

Figure 2.1 demonstrates the portability of `gWidgets` commands, as it shows realizations on different operating systems and with different graphical toolkits.

2.1 Constructors

We begin with some sample `gWidgets` commands that set up a basic interface allowing a user to input some test. The first line loads the package, the others will be described later.

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

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

`gWidgets` most constructors have the following form:

```
gname(arguments, handler = NULL, action = NULL,
```

2. gWIDGETS: OVERVIEW

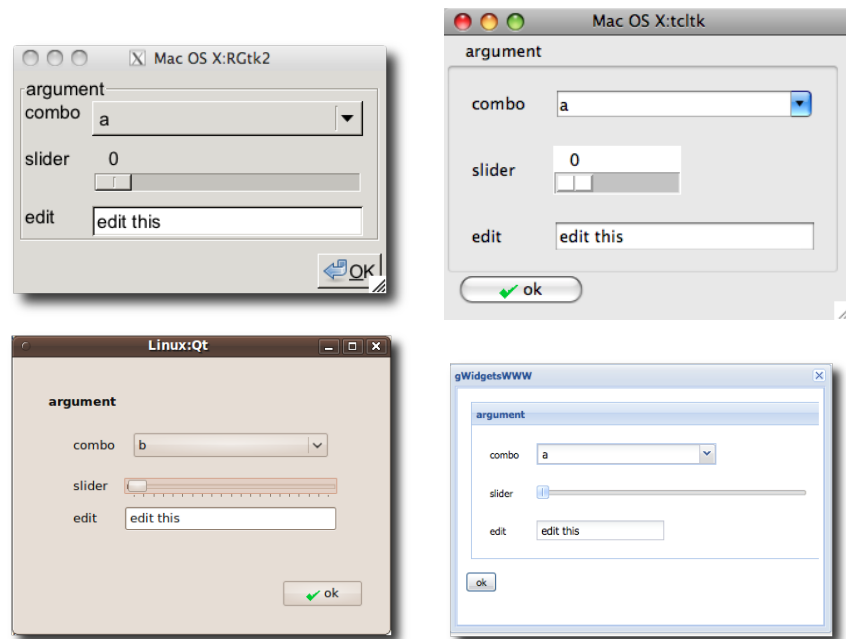


Figure 2.1: The gWidgets package works with different operating systems and different GUI toolkits. This shows, the same code using the RGtk2, tcltk, qtbase packages for a toolkit. Additionally, the gWidgetsWWW package is used in the lower right figure.

```
container = NULL,..., toolkit=guiToolkit())
```

where the arguments vary depending on the object being made.

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

A top-level window does not have a parent container, but other GUI components do. In gWidgets, for the sake of portability, the parent container is passed to the widget constructor through the `container` argument, as it done in all the other constructors. This argument name can always be abbreviated `cont`. This defines the GUI layout, a topic taken up in Chapter 3.

The `toolkit` argument is usually not specified. It is there to allow the user to mix toolkits within the same R session, but in practice can cause problems due to competing event loops. The default call `guiToolkit` which will query the user for a choice of toolkit, if one has not been specified and a choice is available. In our example we have the call

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

to explicitly set the toolkit, thereby avoiding being asked when the `toolkit` argument is evaluated.

The constructors produce three general types of objects: containers, such as the top level window `w` and the box container `g` (Table 3.1); components, such as a label `l`, the edit area `e` or the button `b`. (Table 4 and Table 5) show the basic and compound widgets.) dialogs.

2.2 Methods

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

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

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

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

```
svalue(e)
```

demonstrates the generic method `svalue` that can be used to get or set the main property of the widget. For the object `e`, the main property is the text, for the button and label widgets this property is the label. The `svalue<-` is used to adjust this property programatically. For the selection widgets, there is a natural mapping between vectors or data frames, and the data to be selected. In this case, the user may want the value selected or the index of the selected value. The `index=TRUE` argument may be given to refer to the index, either when getting or setting the properties value. For these selection widgets the `[]` and `[<-` methods refer to the underlying data to be selected from.

¹ In `gWidgets` constructors are generic functions that dispatch down to the toolkit implementation after considering the `toolkit` argument. For example, `gbutton` calls `.gbutton`, defined in the toolkit implementation. This in turn calls a GUI toolkit command, such as `gtkButton`. This is then placed as a slot in an S4 object which in turn is placed in another S4 object, which is the GUI object available to interact with. As such, in `gWidgets`, generic methods also have a double dispatch when called.

The actual class of the S4 object returned by the first constructor is (mostly) not considered, but when we refer to methods for an object, we gloss over this double dispatch and think of it as a single dispatch. This design allows the toolkit packages the freedom to implement their own class structure.

The call,

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

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

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

The methods `tag` and `tag<-` are similar to the base `attr` function. However, the attributes persist across copies of the object. These are implemented to bypass the pass-by-copy issues that can make GUI programming awkward at times.

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

2.3 Callbacks

In our example, the lines

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

create the button object. The argument `handler` is used to bind a callback to the click event of the button. Handlers in `gWidgets` have a common signature `(h,...)` where `h` is a list with components `obj`, to pass in the object of the event (the button in this case), and `action` to pass along any value specified to `action` argument. The latter allows 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 argument is not optional. For some classes, extra information is passed along, for instance for the drop target generic, the component `dropdata` contains a string holding the drag-and-drop information.

While one can specify a callback to the constructor, it is often a better practice to keep separate the construction of an object and the definition of

Table 2.1: Generic functions provided or used in the `gWidgets` API.

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

its callback. The package provides a number of methods to add callbacks for different events. The main method is `addHandlerChanged`, which is used to assign a callback for the typical event for the widget, such as the clicking of a button. (The `handler` argument, when specified, uses this method call.) In addition, there are many "`addHandlerXXX`" methods to assign callbacks to other events, where the `XXX` describes the event. These are useful in the case where more than one event is of interest. For example, for single line text widgets, like `e` in our example, the `addHandlerChanged` method sets a callback to respond when the user finishes editing, whereas a handler set by `addHandlerKeystroke` is called each time a key is pressed. Table 2.3 shows a list of these other methods.

As an example, we could have specified the button as

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

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

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

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.

2.4 Dialogs

The `gWidgets` package provides a few constructors to quickly make some basic dialogs for showing messages or gathering information. Mostly these are modal dialogs that take control of the event loop, not allowing any other part of the GUI to be active for programmatic interaction. As such, constructors of modal dialogs do not return an object to manipulate through its methods, but rather return the user response to the dialog. Hence, they are used differently than other constructors. 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`.

Here we describe the dialogs that can be used to display a message or gather a simple amount of test. The `gfile` dialog is described in Section 4.2 and the `gbasicdialog`, which is implemented like a container, is described in Section 3.1.

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,

Table 2.2: Generic functions to add callbacks in gWidgets API.

Method	Description
<code>addHandlerChanged</code>	Primary handler call for when a widget's value is "changed." The interpretation of "change" depends on the widget.
<code>addHandlerClicked</code>	Sets handler for when widget is clicked with (left) mouse button. May return position of click through components x and y of the h-list.
<code>addHandlerDoubleClick</code>	Sets handler for when widget is double clicked
<code>addHandlerRightclick</code>	Sets handler for when widget is right clicked
<code>addHandlerKeystroke</code>	Sets handler for when key is pressed. The key component is set to this value if possible.
<code>addHandlerFocus</code>	Sets handler for when widget gets focus
<code>addHandlerBlur</code>	Sets handler for when widget loses focus
<code>addHandlerExpose</code>	Sets handler for when widget is first drawn
<code>addHandlerDestroy</code>	Sets handler for when widget is destroyed
<code>addHandlerUnrealize</code>	Sets handler for when widget is undrawn on screen
<code>addHandlerMouseMotion</code>	Sets handler for when widget has mouse go over it
<code>addHandler</code>	For non cross-toolkit use, allows one to specify an underlying signal from the graphical toolkit
<code>removeHandler</code>	Remove a handler from a widget
<code>blockHandler</code>	Temporarily block a handler from being called
<code>unblockHandler</code>	Restore handler that has been blocked
<code>addHandlerIdle</code>	Call a handler during idle time
<code>addPopupMenu</code>	Bind popup menu to widget
<code>add3rdMousePopupMenu</code>	Bind popup menu to right mouse click
<code>addDropSource</code>	Specify a widget as a drop source
<code>addDropMotion</code>	Sets handler to be called when drag event mouses over the widget
<code>addDropTarget</code>	Sets handler to be called on a drop event. Adds the component dropdata.

Table 2.3: Table of constructors for basic dialogs in gWidgets

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

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

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

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

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

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

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

This illustrates how to use the return value.

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

2.5 Installation

The `gWidgets` package started as a port to RGtk2 of the `iWidgets` interface, initially implemented only for Swing through `rJava` (Urbanek). The `gWidgets` package enhances that original interface in terms of functionality and implemented the API for multiple toolkits. As such, an installation requires several different pieces.

The `gWidgets` package is installed as other R packages that reside on CRAN, e.g. through the function `install.packages`. The `gWidgets` package

only provides the application programming interface (API). To actually create a GUI, one needs to also have:

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

Not all features of the API are supported by a particular toolkit. 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. For the most part, the omissions are gracefully handled by simply providing less functionality. We make note of major differences here, realizing that over time they may may be resolved. Consult the package documentation if in doubt.

gWidgets: Layout

GUI construction involves three basic steps: creation and configuration of the main components; the layout of these components; and linking the components to make a GUI interactive. This chapter discusses the layout process within `gWidgets`. Layout is done by placing child components within parent containers which in turn may be nested in other containers. (This is more like GTK+, and not Qt where layout managers control where the components are displayed.) The `gWidgets` package provides a just few useful containers: top-level windows, box containers, grid-like containers and notebook containers. Figure 3.1 shows a paned window, a framed container, a notebook container and box containers used to produce a GUI to show contents of a file.

The primary method for layout is the `add` method for containers. The basic call is of the form `add(parent, child, extra_arguments)`. However, this isn't typically used. In some toolkits, notably `tcltk`, the widget constructors require the specification of a parent window for the widget. To accomodate that, in the `gWidgets` constructors – except for top-level windows – we have the argument `container` (which can always be shortened to `cont`). This is used to specify the immediate parent container, the parent window is found from that. 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 `qtbases` – and the two aspects can be separated – for the sake of cross-toolkit portability we don't illustrate this here.

3.1 Top-level windows

The `gwindow` constructor creates top-level windows. The main window property is the title which is typically displayed at the top of the window. This can be set during construction via the `title` argument or accessed later through the `svalue<-` method. A basic window then is constructed as follows:

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

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

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

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

The optional `visible` argument, used above with its default value `TRUE`¹, controls whether the window is initially drawn. If not drawn, the `visible<-method`, taking a logical value, can be used to draw the window later. Often it is good practice to suppress the initial drawing, especially for displaying GUIs with several controls as the incremental drawing of subsequent child components can make the GUI seem sluggish. As well, this allows the underlying toolkit to compute the necessary size before it is displayed.

Size and placement In GUI programming, a window geometry is a specification of position and size, often abbreviated $w \times h + x + y$. The width and height can be specified at construction through the `width` and `height` arguments. This initial size is the default size, but may be adjusted later through the `size` method or through the window manager.

The initial placement of a window, $x + y$, will be decided by the window manager, unless the `parent` argument is specified. If this is done with a vector of x and y pixel values, the upper left corner will be placed there.

The `parent` argument can also be another `gwindow` instance. In this case, the new window will be positioned over the specified window and be transient for the window. That is, it will be disposed when the parent window is. This is useful, say, when a main window opens a dialog window to gather values.

For example,

```
childw <- gwindow("A child window", parent=w, size=c(200,100))
```

Handlers Windows can be closed programmatically with the `dispose` method. Windows may also be closed through the window manager, by clicking a close icon in the title bar. The `handler` argument is called just before the window is destroyed, but 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,...) {
```

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

Table 3.1: Constructors for container objects

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

```
!gconfirm("Really close", parent=h$obj)
})
```

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

A modal window

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

As with the `gconfirm` dialog, this widget returns `TRUE` or `FALSE` depending on the user's selection. To do something more complicated than `gconfirm`, a handler should be specified at construction. The `handler` argument is used for this. The specified handler is called before the dialog is disposed.

This dialog is used in a slightly different manner, requiring the use of a call to `visible` (not `visible<-`). There are three basic steps: an initial call to `gbasicdialog` to return a container to be used as the parent container for a child component; a construction of the dialog; then a call to the `visible` method on the dialog with `set=TRUE` value (not though `visible(obj) <- TRUE`).

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

```
w <- gbasicdialog("A modal dialog", handler=function(h,...) {
  print(svalue(l))
})
l <- glabel("A simple label", cont=w)
```

3. gWIDGETS: LAYOUT

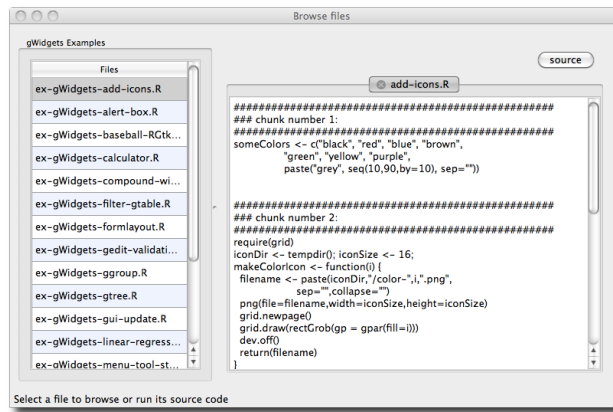


Figure 3.1: The example browser for gWidgets showing different layout components.

Table 3.2: 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

```
visible(w, TRUE) # not visible(w) <- TRUE
```

3.2 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:

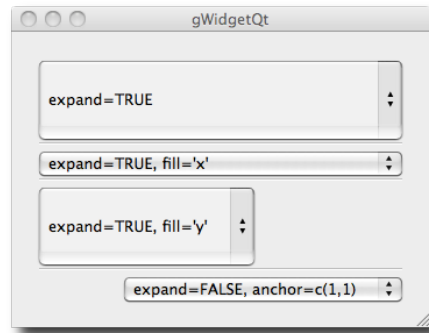


Figure 3.2: Different combinations of `expand`, `fill` and `anchor` for comboboxes in `gWidgetsQt`. The `fill` and `anchor` arguments may be overridden by the underlying toolkit for some widgets.

```
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. Unlike with the underlying graphical toolkits, there is no means to specify other styles of packing such as from the ends, or in the middle by some index.

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

These arguments are:

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

Filling varies from toolkit to toolkit, in particular in `gWidgetsRGtk2` one can really only specify `fill="both"`.

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

Figure 3.2 shows combinations of these arguments under `gWidgetsQt`.

3. GWidgets: LAYOUT

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

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

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

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

To illustrate how the right panel of Figure 3.1 was done, we used nested layouts as follows:

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

Sizing The overall size of a `ggroup` container is controlled through it being a child of its parent container. However, a size can be assigned through the `size<-` method. This will be a preferred size, but need not be the actual size, as the container may need to be drawn larger to accomodate its children.

For `gWidgetsRGtk2` the argument `use.scrollwindow`, when specified as `TRUE`, will add scrollbars to the box container so that a fixed size can be maintained. (Although, it is generally considered a poor idea to use scrollbars when there is a chance the key controls for a dialog will be hidden, this can be useful for displaying lists of data.)

Example 3.1: The `use.scrollwindow` argument

This example shows the `use.scrollwindow` feature for `gWidgetsRGtk2`. The

feature is not well implemented in the other toolkit packages. Widgets that list similar types of data are a common part of GUIs for smart phones, say. The `addItem` function is used to generate a consistent item type, in this case just a label and cancel button. The latter shows the `delete` method for containers.

```
w <- gwindow("Scroll window example", visible=FALSE)
g <- ggroup(cont=w, horizontal=FALSE, use.scrollwindow=TRUE)
addItem <- function(container, i) {
  g1 <- ggroup(cont=container)
  glabel(sprintf("data about %s", i), cont=g1)
  addSpring(g1)
  gimage("cancel", dir="stock", cont=g1, handler=function(h,...) {
    delete(g, g1)
  })
}
for(i in state.name) addItem(g, i)
visible(w) <- TRUE
```

The `gframe` and `gexpandgroup` containers

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

Framed containers are used to set off its child elements using a border and label. The `gframe` constructor produces them. 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.

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.

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

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

The basic framed container is used along these lines:

```
w <- gwindow("gframe example")
f <- gframe("Select a variable:", cont=w)
vars <- gtable(names(mtcars), cont=f, expand=TRUE)
```

The usage of an expanding group is similar. Here we show how one might leave optional the display of a statistical summary of a model.

3. gWIDGETS: LAYOUT

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

3.3 Paned containers: the gpanedgroup container

The `gpanedgroup` constructor produces a container which has two children which are separated by a visual gutter which can be adjusted by the user with their mouse to allocate the space between the two children. Figure 3.1 uses such a container to separate the file selection controls from the file display ones. For this container, the children are aligned side-by-side (by default) or top to bottom if the `horizontal` argument is given as `FALSE`.

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

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

A simplified version of the layout in Figure 3.1 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)
t <- gtext("", cont=pg, expand=TRUE)
visible(w) <- TRUE
svalue(pg) <- 0.33 # after drawing
```

3.4 Tabbed notebooks: the gnotebook container

The `gnotebook` constructor produces a tabbed notebook container. The GUI in Figure 3.1 uses a notebook to hold different text widgets, one for each file being displayed.

The `gWidgets` constructor has a few arguments. The argument `tab.pos` is used to specify the location of the tabs using a value of 1 through 4 with 1 being bottom, 2 left side, 3 top and 4 right side being used, with the default

being 3 (similar numbering is used in `par`). The `closebuttons` argument takes a logical indicating whether the tabs should have close buttons on them. In this case, the argument `dontCloseThese` can be used to specify which tabs, by index, should not be closable. (As of writing, this is not implemented in `gWidgetstcltk`.)

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

The `svalue` method returns the index of the currently raised tab, whereas `svalue<-` can be used to switch the page to the specified tab. The currently shown tab can be removed using the `dispose` method. To remove a different tab, use this method in combination with `svalue<-`. (When removing many tabs, you will want to start from the end as otherwise the tab positions change during removal.)

From some viewpoint, the notebook widget is viewed as a vector with a `names` attribute (the labels) and components being the child components. As such, the `names` method can be used to retrieve the tab names, and `names<-` to set the names. The `length` method returns the number of pages held by the notebook. The `[]` is also implemented to return the child components.

Example 3.2: Tabbed notebook example

In the GUI of Figure 3.1 a notebook is used to hold differing pages. The following is the basic setup used.

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

New pages are added as follows:

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

guiWidget of type: `gTextRGtk` for toolkit: `guiWidgetsToolkitRGtk2`

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)
```

3.5 Grid layout: the `glayout` container

The layout of dialogs and forms is usually seen with some form of alignment between the widgets. The `glayout` constructor provides a grid container to do so, using matrix notation to specify location of the children.

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

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

The constructor has a few arguments to configure the appearance of the container. The spacing between each cell may be specified through the `spacing` argument, the default is 10 pixels. A value of 5 is used above to tighten up the display. To impose a uniform cell size, the `homogeneous` argument can be specified with a value of `TRUE`. The default is `FALSE`.

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

To add a child, the `glayout` container should be specified as the container and be on the left hand side of the `[<-` call. (This is necessary only for the toolkits where a container must be specified, where the right hand side is used to pass along the parent information and the left hand side is used for the layout.) For convenience, if the right hand side is a string, a label will be generated. To align a widget within a cell, the `anchor` argument of the `[<-glayout` method is used. The example above illustrates how this can be used to achieve a center balance.

gWidgets: Control Widgets

This section discusses the basic controls provided by gWidgets.

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 4) an icon will appear. (The ok button below, but not the custom par... one.)

In common with the other controls, the argument handler is used to specify a callback and the action argument will be passed along to this callback (unless it is a gaction object, whose case is described in Section 4.5). 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...", cont=bg)
addHandlerClicked(parButton, handler=function(h,...) {
  w1 <- gwindow("Set par values...", 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))
  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)
```

4. gWIDGETS: CONTROL WIDGETS

Table 4.1: Table of constructors for control widgets in gWidgets. Most, but not all, are implemented for each toolkit.

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

```
        dispose(w1)
      })
    })
```

The button's label is its main property and can be queried or set with `The $value` or `$value<-`. Most GUIs will make a button insensitive to user input if the button's action is not currently permissible. Toolkits draw such buttons in a grayed-out state. As with other components, the `enabled<-` method can set or disable whether a widget can accept input.

Labels

The `glabel` constructor produces a basic label widget. We've 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 4.1). For `gWidgetsRGtk2` the argument `markup` for the constructor takes a logical value indicating if the text is in the native markup language (PANGO).

For example, to make a form's labels have some emphasis we could do:

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

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

Statusbars

Statusbars are simply labels placed at the bottom of a top-level window to leave informative, but non-disruptive, messages for the user. The `gstatusbar` constructor provides this widget. The `container` argument should be a top-level window instance. The only property is the label's text. This may be specified at construction with the argument `text`. Subsequent changes are made through the `svalue<-` method.

Displaying icons and images stored in files

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

Other graphic files and the stock icons can be displayed by the `gimage` widget.¹ The file to display is specified through the `filename` argument of

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

the constructor. This value is combined with that of the `dirname` argument to specify the file path. Stock icons, can be specified by using their name for the `filename` argument and the character string "stock" for the `dirname` argument.²

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

The default handler is a button click handler.

To illustrate, a simple means to display a graphic (except in `gWidgetstcltk` where png support is not available) is as follows:

```
f <- tempfile()
png(f)
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 an example.

Example 4.1: Adding and using stock icons

This example shows how to add to the available stock icons and use `gimage` to display them. It creates a table to select a color from, as an alternative to a more complicated color chooser dialog. Under `gWidgetstcltk` the image files would need to be converted to gif format, as png format is not a natively supported image type.

We begin by defining 16 arbitrary colors.

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

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

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

²For `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".


```
}
```

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

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

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

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

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 only available for `gWidgetsQt`.

4.1 Text editing controls

Single-line, editable text

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. For some toolkits, a `ghtml` widget is also defined, but neither `gWidgetsRGtk2` nor `gWidgetstcltk` have this implemented.

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

Methods The text is returned by the `svalue` method and may be set through the `svalue<-` method. The `svalue` method will return a character vector by default. However, it may be desirable to use this widget to collect numeric values or perhaps some other type of variable. One could write code to coerce

4. gWIDGETS: CONTROL WIDGETS

the character to the desired type, but it is sometimes convenient to have the return value be a certain non-character type. In this case, the `coerce.with` argument can be used to specify a function of a single argument to call before the value is returned by `svalue`.

Some toolkits allow type-ahead values to be set. These values anticipate what a user wishes to type and offers a means to complete a word. The `[<-` method allows these values to be 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")
g <- ggroup(cont=w)
glabel("State name:", cont=g)
```

guiWidget of type: gLabelRGtk for toolkit: guiWidgetsToolkitRGtk2

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

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

Example 4.2: Validation

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

The `coerce.with` argument can be used to specify a function to coerce values after an action is initiated, but in this example we show how to validate the text widget when it loses focus. If the value is invalid, we set the text color to red. In general, validation is much better implemented by the underlying toolkits, especially Qt.

This regular expression is used to validate against:

```
validRegexpr <- "[[:digit:]]{3}-[[:digit:]]{4}"
isValid <- function(val)
  grepl(validRegexpr, val)
```

Our basic GUI follows.

```
w <- gwindow("Validation example")
tbl <- glayout(cont=w)
tbl[1,1] <- "Phone number (XXX-XXXX)"
tbl[1,2] <- (e <- gedit("", cont = tbl))
tbl[2,2] <- (b <- gbutton("submit", cont = tbl,
                          handler=function(h,...) print("hi")))
```

We validate by checking the widget's current value against the regular expression when the widget loses focus. For this, the `blur` event is a focus-out event. Alternatively, one may also want to use `addHandlerKeystroke`, to validate after each key press, or `addHandlerChanged`, to validate when the value is committed (typically when the user presses the enter key).

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

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, which is useful with `gWidgetstcltk`.

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 can 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”.

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

fonts Fonts can be specified for the entire buffer or the selection using the specifications in Table 4.1. To specify fonts for the entire buffer use the `font.attr` argument of the constructor. The `font<-` method serves the same purpose, provided there is no selection when called. If there is a selection, the font change will only be applied to the selection. Finally, the `font.attr`

Table 4.2: 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

argument for the `insert` method specifies the font attributes for the inserted text.

Example 4.3: A calculator

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

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

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

```
buttons <- rbind(c(7:9, "(", ")"),
                c(4:6, "*", "/"),
                c(1:3, "+", "-"))

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

```
outputArea <- gtext("", cont = g)
```

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

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

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

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

4.2 Selection controls

A common task for a GUI control is to select a value or values from a set of numbers or a table of numbers. Many toolkits implement these widgets using a model-view-controller paradigm whereby the control is just one of possibly many views of the data model. This approach is not used by `gWidgets`. Rather, each widget has its own data store (like a vector or data frame) containing the data for selection, and 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.

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

Checkbox widget

The simplest selection control is the checkbox widget that allows the user to set a state as `TRUE` or `FALSE`. The constructor has an argument `text` to set a

label and checked to indicate if the widget should initially be checked. The default is TRUE.

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.

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

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

A group of checkboxes

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

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

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

As an illustration of the related selection widgets, the following could be part of a GUI to illustrate densities for various kernels.

```
kerns <- as.character(formals(density.default)$kernel)[-1]
w <- gwindow("Arguments for density example")
lyt <- glayout(cont=w)
lyt[1,1] <- "bw"
lyt[1,2, anchor=c(-1,0)] <- gradio(c("nrd0", "nrd", "ucv", "bcv", "SJ"), cont=lyt)
lyt[2,1] <- "Kernel"
lyt[2,2, anchor=c(-1,0)] <- gcheckboxgroup(kerns, cont=lyt)
lyt[3,1] <- "na.rm"
lyt[3,2, anchor=c(-1,0)] <- gcheckbox("na.rm", checked=TRUE, cont=lyt)
lyt[4,2] <- gbutton("done", cont=lyt, handler=function(h,...) {
  out <- sapply(1:3, function(i) {
    widget <- lyt[i,2]
    svalue(widget)
  })
  print(out)                                # make a plot ...
})
```

A combobox

A combobox is used as an alternative to a radio button group when there are too many choices to comfortably fit on the screen. Comboboxes are constructed by `gcombobox`. The possible choices are specified to the argument `items`. This may be a vector of values or a data frame whose first column defines the choices. For toolkits which support icons in the combobox, if the data is specified as a data frame, the second column signifies which stock icon is to be used. By design, a third column specifies a tooltip to appear when the mouse hovers over a selection, but this is only implemented for `gWidgetsQt`.

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

4. gWIDGETS: CONTROL WIDGETS

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

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 state can be set through the `svalue<-` method. This is specified by a character unless `index` is `TRUE`, in which case as a numeric index with respect to the underlying items. The `[]` method returns the items of the data store, and `[-` is used to assign new values to the data store. The value may be a vector, or data frame if an icon or tooltip is being assigned. The `length` method returns the number of items.

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

A slider control

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

The `svalue` method returns the currently chosen value. The `[-` method can be used to update the sequence of values to choose from. The new assignment should be a regularly spaced sequence of numbers, as returned by `seq`.

The default handler is called when the slider is changed. Example 4.4 shows how this can be used to update a graphic.

A spin button control

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

Example 4.4: Example of sliders and spin buttons

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

```
w <- gwindow("Slider and Spin Button example")
tbl <- glayout(cont=w)
tbl[1,1] <- "sample size"
tbl[1,2] <- (spinner <- gspinbutton(from=10, to=100, by=5,
                                   value=25, cont=tbl))

tbl[2,1] <- "adjusted bandwidth"
tbl[2,2, expand=TRUE] <- (slider <- gslider(from=0.1, to=1,
                                             by=0.01, value=1, cont=tbl))
plotGraph <- function(h,...) {
  x <- rexp(svalue(spinner))
  plot(density(x, adj=svalue(slider)))
}
sapply(list(spinner, slider), function(i)
  addHandlerChanged(i, handler=plotGraph))
```

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

```
if(!is.na(tmp <- gfile()))
  source(tmp)
## or
gfile(handler=function(h,...) {
  if(!is.na(h$file))
    source(h$file)
})
```

Selecting a date

The `gcalendar` constructor returns a widget for selecting a date, if there is a native widget in the underlying toolkit, or a text edit box for entering a date. The argument `text` 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.

4.3 Table and tree controls

XXX table and tree separated off – why?

Display of tabular data

The `gtable` constructor produces a widget that displays data in a tabular form from which the user can select one (or more) rows. The widgets performance under `gWidgetsRGtk2` and `gWidgetsQt` is much faster and able to handle larger data stores than under `gWidgetsTclTk`, as there is no native table widget in `Tcl/Tk`. All perform well on moderate-sized data sets (10 or so columns and fewer than 500 rows),

The data is specified through the `items` argument. This 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.

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.

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

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

Selection Users can select a row, not a cell from this widget. The value returned by a selection is controlled by the constructor's arguments `chosencol`, which specifies which column value will be returned, as the user can only

specify the row; and `multiple` which controls whether the user may select more than one row.

Methods The `svalue` method will return the currently selected value. If the argument `index` is specified as `TRUE`, then the selected row index (or indices) will be returned. These refer to the data store, not the visible data when filtering is being used (below). The argument `drop` specifies if just the chosen column's value is returned (the default) or, if specified as `FALSE`, the entire row.

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

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

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

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

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

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

Filtering The arguments `filter.column` and `filter.FUN` allow one to specify whether the user can filter, or limit, the display of the values in the data store. If a column number is specified to `filter.column` then a combobox is added to the widget with values taken from the unique values in the specified column. Changing the value of the combobox restricts the display of the data

to just those rows where the value in the filter column matches the combobox 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 combobox whose values are specified through the argument `filter.labels`. This function should return a logical vector with length matching the number of rows in the data frame. Only rows corresponding to TRUE values will be displayed. If `filter.FUN` is the character string "manual" then the `visible<-` method can be used to control the filtering, again by specifying a logical vector of the proper length. See Example 4.6 for an application.

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

Example 4.5: Simple filtering

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

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

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

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

Example 4.6: More complex filtering

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

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

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

This basic GUI is barebones, for example we skip adding text labels to guide the user.

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

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

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

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

Finally, the `addHandlerKeystroke` method calls its handler everytime a key is released while the focus is in the edit widget. In this case, the handler finds the matching indices using the `ourMatch` function, converts these into logical format, and then updates the display using the `visible<-` method for `gtable`.

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

4.4 Display of heirarchical data

The `gtree` constructor can be used to display heirarchical structures, such as a file system. This constructor parameterizes the data to be displayed in terms of the node of the tree that is currently selected. The `offspring` argument is assigned a function of two arguments, the path a particular node and

the arbitrary object passed through the optional `offspring.data` argument. This function should return a data frame with each row referring to an offspring for the node and whose first column is a key that identifies each of the offspring, unless the argument `chosencol` is used to specify otherwise.

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

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

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

The above will produce a tree with just one column. 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="11", b="12", c=list(a="111")))
o <- listOffspring(l)
w <- gwindow("Tree test")
t <- gtree(o, cont=w)
```

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

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

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

Methods The `svalue` method returns the currently selected key, or node label. There is no assignment method. The `[]` method returns the path for the currently selected node. This is what is passed to the `offspring` function. The `update` method updates the displayed tree. The method `addHandlerDoubleClick` specifies a function to call on a double click event.

Example 4.7: Using `gtree` to explore a recursive partition

The `party` package implements a recursive partitioning algorithm for tree-based regression and classification models. The package provides an excellent plot method for the object, but in this example we demonstrate how the `gtree` widget can be used to display the heirarchical 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 heirarchical nature through its nodes. This object has a complex structure using lists to store data about the nodes. We define an `offspring` function next that tracks the node by number, as is done in the `party` object; records whether a node has offspring through the terminal component (bypassing the `hasOffspring` function); and computes a condition on the variable that creates the node. For this example, the trees are all binary trees with 0 or 2 offspring so this data frame has only 0 or 2 rows.

```
offspring <- function(key, offspring.data) {
  if(missing(key) || length(key) == 0) # which node?
    node <- 1
  else
    node <- as.numeric(key[length(key)]) # key is a vector

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

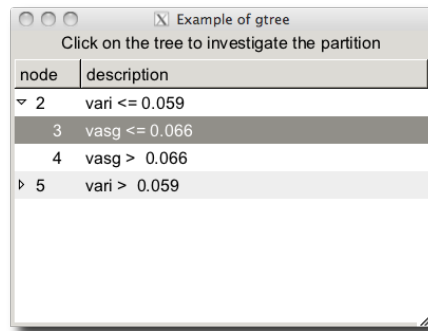


Figure 4.1: GUI to explore return value of a model fit by the party package.

```
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 4.1)

```
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)
```

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, ])
```



```
}})
```

4.5 Actions, menus and toolbars

XXX

Actions

Actions are invisible objects representing an application command that is executable through one or more widgets. See ?? for more details. Actions in `gWidgets` are created through the `gaction` constructor. The arguments are `label`, `tooltip`, `icon`, `key.accel` and the standard handler and action. The label appears as the text on a button, the menu item or toolbar text, whereas the icon will decorate the same if possible. For some toolkits, the tooltip pops up when the mouse hovers. (See also the `tooltip<-` method.)

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

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

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

Toolbars

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

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

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.) The argument `style` can be one of `"both"`, `"icons"`, `"text"`, or `"both-horiz"` to specify

how the toolbar is rendered. Toolbars in `gWidgetstcltk` are not native widgets, so the implementation uses aligned buttons.

Menubars, popup menus

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

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

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

Example 4.8: Menubar and toolbar example

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

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

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

```
t1 <- list(open = aOpen, quit = aQuit)
m1 <- list(File = list(
  open = aOpen,
  sep = gseparator(),
  quit = aQuit),
  Edit = list(
    undo = aUndo
  ))
```

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

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

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

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

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

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

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

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

Popup menus Popup menus can be created for a right click event through the `add3rdMousePopupMenu` constructor. (Or `control-button-1` for Mac OS X.) This constructor has arguments `obj` to specify a widget, like a button, to initiate the popup, `menulist` to specify the menu and optionally an action argument.

Example 4.9: Popup menus

```
w <- gwindow("Popup example")
b <- gbutton("click me or right click me", cont=w,
  handler=function(h, ...) {
    cat("You clicked me\n")
  })
f <- function(h,...) cat("you right clicked on", h$action)
mbList <- list(one = gaction("one", action="one", handler=f),
  two = gaction("two", action="two", handler=f)
)
add3rdMousePopupMenu(b, mbList)
```


gWidgets: R-specific widgets

The `gWidgets` package provides some R specific widgets for producing GUIs. Table 5 lists them.

5.1 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`, `ps` are similar to other graphics devices.

When working with multiple devices, it becomes necessary to switch between devices. A `ggraphics` instance can be made to represent the current device if the user clicks in the window. Otherwise, the `visible<-` method can be used to set the object as the current device.

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

```
w <- gwindow("ggraphics example")
g <- ggraphics(cont=w)
```

Table 5.1: Table of constructors for compound widgets in `gWidgets`

Constructor	Description
<code>gvarbrowser</code>	GUI for browsing variables in the workspace
<code>gcommandline</code>	Command line widget
<code>gformlayout</code>	Creates a GUI from a list specifying layout
<code>ggenericwidget</code>	Creates a GUI for a function based on its formal arguments or a defining list

5. gWIDGETS: R-SPECIFIC WIDGETS

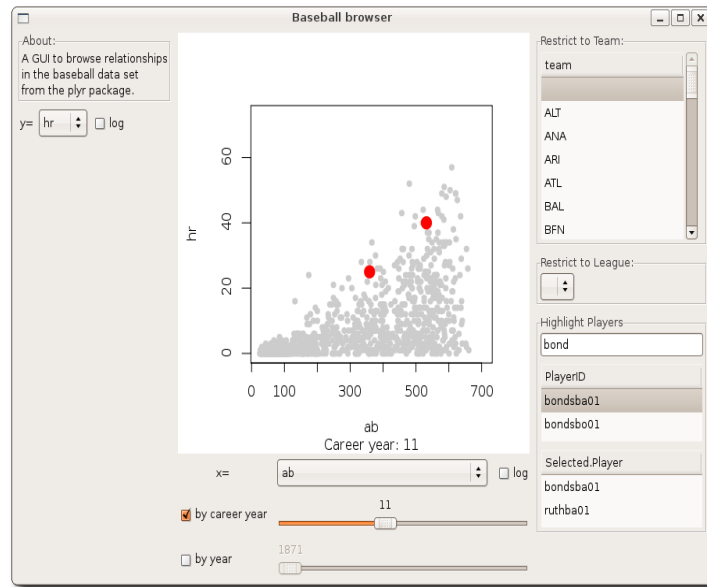


Figure 5.1: A RGtk2 GUI for exploring the baseball data set of the plyr package. One can subset by year or career year through the slider widgets.

```
x <- mtcars$wt; y <- mtcars$mpg
#
addHandlerClicked(g, handler=function(h,...) {
  cat(sprintf("You clicked %s x %s\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]))
  }
})
#
plot(x, y)
```

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

Example 5.1: A GUI to explore a data set

This example creates a GUI to explore the baseball data set of the plyr

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

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

This example uses environments to store data and our widgets.

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

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

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

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

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

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

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

5. gWIDGETS: R-SPECIFIC WIDGETS

```
f <- gframe("About:", cont=lp)
l <- glabel(paste("A GUI to browse relationships",
  "in the baseball data set",
  "from the plyr package.",
  sep="\n"),
  cont=f)
g1 <- ggroup(cont=lp)
l <- glabel("y=", cont=g1)
e$y <- gcombobox(nVars, selected=4, cont=g1)
e$ylog <- gcheckbox("log", checked=FALSE, cont=g1)
```

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

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

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

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



```

size(e$team) <- c(200,200)
svalue(e$team, index=TRUE) <- 1
##
f <- gframe("Restrict to League:", cont=rp)
leagues <- names(table(b$lg))[-1]      # drop ""
e$lg <- gcombobox(c("", leagues), cont=f)
##
f <- gframe("Highlight Players", horizontal=FALSE, cont=rp)
searchPlayer <- gedit("", cont=f)
listPlayers <- gtable(data.frame("PlayerID"=unique(b$id),
                                stringsAsFactors=FALSE),
                      filter.FUN="manual", cont=f)
e$id <- gtable(data.frame("Selected Player"=character(0),
                          stringsAsFactors=FALSE), cont=f)

```

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

```

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

```

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

```

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

```

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

```

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

```

These handlers are used to select the IDs to highlight.

```

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

```

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

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

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

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

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

The `traitr` package, an add on for `gWidgets`, can simplify the construction of such GUIs. The package vignette provides an example.

5.2 A data frame editor

The `gdf` constructor returns a widget for editing data frames. This is similar to the GUI provided by the `data.entry` function, but uses the underlying

toolkit in use by `gWidgets`. The implementations differ between toolkits, with some offering much more. We describe what is in common below.¹

Each cell can be edited. Users can click (or double click) in a cell to select it, or use the arrow and tab keys to navigate. The constructor has argument `items` to specify the data frame to edit and `name` to specify the data frame name, if desired. The column types are important, in particular factors and character types are treated differently, although they may render in a similar manner.

Methods There are several methods defined that follow those of a data frame. The `[` and `[<-` methods can be used to extract and set values from the data frame by index. As with `gtable`, these are not as flexible as for a data frame. 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 `obj[,]` will return the current data frame, which can be assigned to some value for saving. The current data frame can be completely replaced, when no indices are specified in the replacement call. Additionally, the data frame methods `dimnames`, `dimnames<-`, `names`, `names<-`, and `length` are defined.

The following methods can be used to assign handlers (all are implemented in `gWidgetsRGtk2`, but not for the others): `addHandlerChanged` (cell changed), `addHandlerClicked`, `addHandlerDoubleClick`, `addHandlerColumnClicked`, `addHandlerColumnDoubleClick`, and `addHandlerColumnRightclick`.

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

Example 5.2: 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`. The GUI will have an embedded graphics device, a visual display of the filtered data, and a means to filter, or narrow, the data that is under consideration. Although, our example is not too feature rich, it illustrates a framework that can easily be extended.

The `gWidgets` package provides a programming interface (an API) but the link between R and the graphical toolkit is provided by other packages. The setting of the `guiToolkit` below, when loading `gWidgets`, will cause the

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. For `gWidgetsRGtk2` there is also the `gdfedit` constructor which is faster and has better usability features. This merely wraps the `gtkDfEdit` function from `RGtk2Extras`. This function is not exported by `gWidgets`, so the toolkit package must be loaded. Again, as with `gtable`, the widget under `gWidgetstcltk` is slower, can load a moderately sized data frame in a reasonable time.

5. gWIDGETS: R-SPECIFIC WIDGETS

RGtk2 package to be loaded, and the GTK+ toolkit to be used.

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
```

A GUI is constructed out of widgets that are layed out within containers, which may be nested. In `gWidgets`, the construction of the widget or container and the layout are combined. In this first code snippet, a notebook container is placed within a toplevel window. The specification is done through the `container` argument, which we abbreviate to `cont`. This argument in turn is passed to the containers `add` method.

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

The `gwindow` argument `visible=FALSE` suppresses the drawing of the top-level window, and is used here, as it gives a snappier appearance when the window is being drawn.

Our notebook widget will have two pages. The first is for a general description of the application. In this example, we use a `glabel` widget to display a message. When we add this widget to the notebook container, the extra argument `label="About"` is passed to the `add` method of the notebook, and is used to create the tab label.

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

A more detailed description would be warranted in an actual application.

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 to box containers that will use a vertical layout (when `horizontal=FALSE`).

```
g <- ggroup(cont=nb, label=gettext("Explore..."))
lg <- ggroup(cont=g, horizontal=FALSE) # vertical boxes
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. This widget is unfortunately not available for all toolkits.

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

`guiWidget` of type: `gGraphicsRGtk` for toolkit: `guiWidgetsToolkitRGtk2`

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. This allows a label to clarify for the user what these controls do.

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

We use a grid layout to arrange our filter widgets, as it just looks nicer. The `gWidgets` `gLayout` container provides this layout with a familiar interface – matrix notation indexing. In this example we store the widgets in a list, `l`, which facilitates the processing of their values later on.

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

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

Adding interactivity The widget construction and their layout is now complete for this example. At this point, we need to make the GUI responsive to user interaction. To that matter, we create a few functions. This first to update the data frame when the filter controls are changed. For this we need to compute a logical variable indicating which rows are to be considered. Within the definition of the function, we use the global variables `l`, `tbl` and `noCases`.

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

5. gWIDGETS: R-SPECIFIC WIDGETS

```
visible(tbl) <- ind                                # ud pate table

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

The `visible<-` and `svalue<-` methods change the underlying widgets. The generic `svalue<-` is used to change the primary value for a widget (and `svalue` returns this value). In the above, we see these methods used to get the values from the comboboxes and to set the text in the label. The `visible<-` method is another generic. In this example it is used to specify which rows of the data are actually displayed by the widget.

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

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

We now add a handler to be called whenever one of our comboboxes is changed. The `gWidgets` package provides a number of methods to call handlers for specific events, but the generic `addHandlerChanged` is meant to be for the most common use.

```
f <- function(...) {
  updateDataFrame()
  updateGraphic()
}
sapply(1, function(i) addHandlerChanged(i, handler=f))
```

Type.changed	Cylinders.changed
2609	2610

For the data display, we wish to allow the user to view cases by clicking on a row. The following will do so. Note the use of `h$obj` below, which within a handler refers to the widget that the event was initiated on. Passing this in to the handler can be useful is one wishes to avoid using global variables.

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

To illustrate some shortcomings of `gWidgets`, there is no means to select points in the graphic and have the display of the data highlighted.

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

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

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, if applicable.

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 name of the currently selected value is returned by the `svalue` method.

The update method will update the list of items being displayed. As, R has no means to notify observers if its global workspace exists, the user can either invoke this method, or have the GUI poll every so often to query for changes (`addHandlerIdle`). As the latter can be time consuming if there are many objects in the global environment, it isn't implemented in for each toolkit.

Example 5.3: 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)
```

guiWidget of type: `gGraphicsRGtk` for toolkit: `guiWidgetsToolkitRGtk2`

```
xlabel <- glabel("", cont=g1)
ylabel <- glabel("", cont=g1)
clear <- gbutton("clear", cont=g1)
```

We create a function to initialize the interface.

5. gWIDGETS: R-SPECIFIC WIDGETS

```
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=function(h,...) {
  initUI()
})
```

Next, we write a function to update the user interface. In this case we need to figure out which state the GUI is currently in by considering the text in each drop 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 callback argument, `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 has the 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 an alternative.

To add a help page, the `add` method is used, where the `value` argument describes the desired page. This can be a character string containing the topic, a character string of the form `package::topic` to specify the package, or a list with named components `package` and `topic`. The `dispose` method of notebooks can be used to remove the current tab.

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

Command line widget

A simple command line widget is created by the `gcommandline` constructor. This is not meant as a replacement for any of R's commandlines, but is provided for lightweight 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,

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

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.

Laying out a form

The `gformlayout` constructor takes a list defining a layout and creates the specified widgets. The design borrows from the `ext.js` javascript libraries for web programming, where a similar function can be used to specify the layout of web forms. Several toolkits have a means to specify a layout using XML (eg. GTK+ Builder and Qt Assistant); this implementation uses a list, under the assumption that it is more familiar to the R user. By defining the layout ahead of time, pieces of the layout can be recycled for other layouts.

A simple example would be

```
l <- list(type="ggroup",
         horizontal=TRUE,
         children=list(
           list(type="glabel",
                text="x:"),
           list(name="x",
                label="asdfas",
                type="gedit",
                text="initial text"),
           list(type="glabel",
                text="state:"),
           list(name="y",
                type="gcombobox",
                items=state.name)
         )
w <- gwindow("glayout example")
f <- gformlayout(l, cont=w)
```

To define the layout, each list element is specified using a named. The `type` element indicates the component to be created, as a string. This can be the name of a container constructor, a widget constructor or the special value `"fieldset"`. Field sets are used to group a set of common controls.

The list defining each GUI component has named elements to pass to the constructor, such as `text` and `items` in the above example, and named elements used by the `gformlayout` constructor. For example, the `name` element when specified, allows that component to be referenced through `svalue`, which returns the form's values in a list, or `l`, which returns the components in a list.

If the type is a container or fieldset, then the `children` element is a list whose elements specify the children as above. Except for fieldsets, these chil-

dren can contain other containers or controls. Fieldsets only allow controls as children.

For fieldsets the `label` element adds a descriptive label to the layout. The `label.pos` element controls the placement of the label. The value "top" places the label on top of the widget, while "side", the default, puts it on the side. The `label.font` element specifies the font properties of the label, as with the `font<-` method.

Parts of the form can be made to depend on other parts. For example, whether a component is enabled or not may be controlled by the `depends.on`, `depends.FUN`, and `depends.signal` elements. If the `depends.on` element specifies the name of a previous component, then the function `depends.FUN` will be consulted when the signal specified by `depends.signal` is emitted. This uses the `addHandlerXXX` names with a default value of `addHandlerChanged`. The `depends.FUN` function should take a single argument consisting of the value returned by `svalue` when called on the widget specified through `depends.on`. This function should return a logical indicating if the widget is enabled or not.

The constructor returns an object with just a few methods. In addition to `svalue` and `[],` the `names` method returns the names of the widgets in the list.

Example 5.4: The `gformlayout` constructor

This example uses `gformlayout` to make a GUI for a *t*-test (Figure 5.2). The first task is to define the list that will set up the GUI. We do this in pieces. This first piece will define the part of the GUI where the null and alternative hypotheses are specified. The null is specified as a numeric value with a default of 0. We use the `gedit` widget which by default will return a character value, so the `coerce.with` argument is specified. For the alternative, this requires a selection for just 3 possibilities, so a combo box is employed.

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

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

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

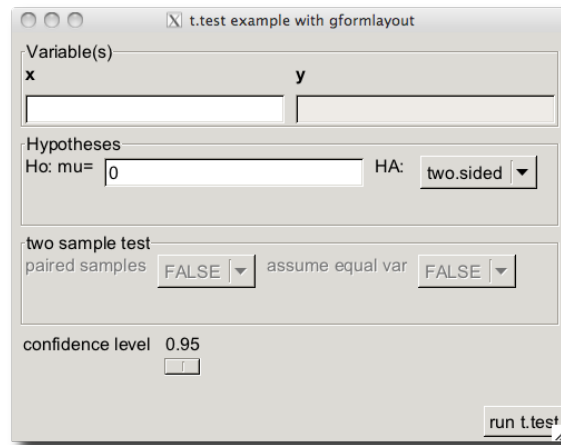


Figure 5.2: A dialog to collect arguments for a t -test made with `gformlayout`.

the `x` value is specified.

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

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

```
two.sample <-
  list(type = "fieldset",
        label = "two sample test",
        columns = 2,
        depends.on = "y",
        depends.FUN = function(value) nchar(value) > 0,
```

```

depends.signal = "addHandlerChanged",
children = list(
  list(type = "gcombobox",
        name = "paired", label = "paired samples",
        items = c(FALSE, TRUE)
  ),
  list(type = "gcombobox",
        name = "var.equal", label = "assume equal var",
        items = c(FALSE, TRUE)
  ))

```

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

```

conf.level <-
  list(type = "fieldset",
        columns = 1,
        children = list(
          list(type = "gslider",
                name = "conf.level", label = "confidence level",
                from=0.5, to=1.0, by=.01, value=0.95
          ))

```

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

```

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

```

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

```

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

```

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

```

addHandlerChanged(b, function(h, ...) {
  out <- svalue(fl)
  out$x <- svalue(out$x) # turns text string into numbers

```

```
if(out$y == "") {
  out$y <- out$paired <- NULL
} else {
  out$y <- svalue(out$y)
}
print(do.call("t.test", out))
})
```

Creating a GUI for a function

The `ggenericwidget` constructor creates a GUI for invoking a given function. The GUI is derived from the formal arguments. The `fgui` package provides a similar function, with some more features, although limited to the `tcltk` toolkit.

The usage is straightforward. To make a GUI for a function is as simple as:

```
f <- function(x=1, variable="a") {
  print("Something with x and variable")
}
g <- ggenericwidget(f, cont=gwindow())
```

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

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

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

```
tmp <- tempfile()
cat(gWidgets::autogenerategeneric(f), file=tmp)
## ... do some edits ...
source(tmp)
w <- gwindow("Another ggeneric widget example")
ggenericwidget(f.LIST, cont=w) # made by autogenerategeneric
```

RGtk2: Overview

As the name implies, the RGtk2 package is an interface, or binding, between R and GTK+, a mature, cross-platform GUI toolkit. The letters *GTK* stand for the *GIMP ToolKit*, with the word *GIMP* recording the origin of the library as part of the GNU Image Manipulation Program. GTK+ provides the same widgets on every platform, though it can be customized to emulate platform-specific look and feel. The library is written in C, which facilitates access from languages like R that are also implemented in C. GTK+ is licensed under the *Lesser GNU Public License* (LGPL), while RGtk2 is under the *GNU Public License* (GPL). The package is available from the Comprehensive R Archive Network (CRAN) at <http://CRAN.R-project.org/package=RGtk2>.

The name RGtk2 also implies that there exists a package named RGtk, which is indeed the case. The original RGtk is bound to the previous generation of GTK+, version 1.2. RGtk2 is based on GTK+ 2.0, the current generation. This book covers RGtk2 specifically, although many of the fundamental features of RGtk2 are inherited from RGtk.

RGtk2 provides virtually all of the functionality in GTK+ to the R programmer. In addition, RGtk2 interfaces with several other libraries in the GTK+ stack: Pango for font rendering; Cairo for vector graphics; GdkPixbuf for image manipulation; libglade for designing GUI layouts from an XML description; ATK for accessible interfaces; and GDK, an abstraction over the native windowing system, supporting either X11 or Windows. These libraries are multi-platform and extensive and have been used for many major projects, such as the Linux versions of Firefox and Open Office.

The API of each of these libraries is mapped to R in a way that is consistent with R conventions and familiar to the R user. Much of the RGtk2 API consists of autogenerated R functions that call into the one of the underlying libraries. For example, the R function `gtkContainerAdd` eventually calls the C function `gtk_container_add`. The naming convention is that the C name has its underscores removed and each following letter capitalized (camelback style).

The full API for GTK+ is quite large, and complete documentation of

it is beyond our scope. However, the GTK+ documentation is algorithmically converted into the R help format during the generation of RGtk2. This conveniently allows the programmer to refer to the appropriate documentation within an R session, without having to consult a web page, such as <http://library.gnome.org/devel/gtk/stable/>, which lists the C API of the stable versions GTK+.

In this chapter, we give an overview of how RGtk2 maps the GTK+ API, including its classes, constructors, methods, properties, signals and enumerations, to an R-level API that is relatively familiar to and convenient for an R user. A simple GUI will be gradually constructed to demonstrate the API.

6.1 Objects and Classes

In any toolkit, all widget types have functionality in common. For example, they are all drawn on the screen in a consistent style. They can be hidden and shown again. To formalize this relationship and to simplify implementation by sharing code between widgets, GTK+, like many other toolkits, defines an inheritance hierarchy for its widget types. In the parlance of object-oriented programming, each type is represented by a *class*.

For specifying the hierarchy, GTK+ relies on GObject, a C library that implements a class-based, single-inheritance object-oriented system. A GObject class encapsulates behaviors that all instances of the class share. Every class has at most one parent from which it inherits the behaviors of its ancestors. A subclass can override some specific inherited behaviors. The interface defined by a class consists of constructors, methods, properties, and signals.

Single inheritance can be restrictive when a class performs multiple roles in a program. To circumvent this, GTK+ adopts the popular concept of the *interface*, which is essentially a contract that specifies which methods, properties and signals a class must implement. As with languages like Java and C#, a class can *implement* multiple interfaces, and an interface can be composed of other interfaces. An interface allows the programmer to treat all instances of implementing classes in a similar way. However, unlike class inheritance, the implementation of the methods, properties and signals is not shared.

We explain the constructors, methods, properties and signals of classes and interfaces in the following sections and demonstrate them in the construction of a simple “Hello World” GUI, shown in Figure 6.1. A more detailed and technical explanation of GObject is available in Section ??.

Figure 6.1: “Hello World” in GTK+. A window containing a single button displaying a label with the text Hello World.

6.2 Constructors

The first step in our example is to create a top-level window to contain our GUI. Creating an instance of a GTK widget requires calling a single R function, known as a constructor. Following R conventions, the constructor for a class has the same name as the class, except the first character is lowercase. The following statement constructs an instance of the `GtkWindow` class:

```
window <- gtkWindow("toplevel", show = FALSE)
```

The first argument to the constructor for `GtkWindow` instructs the window manager to treat the window as top-level. The `show` argument is the last argument for every widget constructor. It indicates whether the widget should be made visible immediately after construction. The default value of `show` is `TRUE`. In this case we want to defer showing the window until after we finish constructing our simple GUI.

At the GTK+ level, a class usually has multiple constructors, each implemented as a separate C function. In RGtk2, the names of these functions all end with `New`. The “meta” constructor `gtkWindow`, called above, automatically delegates to one of the low-level constructors, based on the provided arguments.

A GTK+ object created by the R user has an R-level object as its proxy. Thus, `window` is a reference to a `GtkWindow` instance. The class hierarchy of a proxy object is represented by the `class` attribute. One interprets the attribute according to S3 conventions, so that the class names are in order from most to least derived:

```
class(window)

[1] "GtkWindow"      "GtkBin"          "GtkContainer"
[4] "GtkWidget"      "GtkObject"       "GInitiallyUnowned"
[7] "GObject"        "RGtkObject"
```

We find that the `GtkWindow` class inherits methods, properties, and signals from the `GtkBin`, `GtkContainer`, `GtkWidget`, `GtkObject`, `GInitiallyUnowned`, and `GObject` classes. Every type of GTK+ widget inherits from the base `GtkWidget` class, which implements the general characteristics shared by all widget classes, e.g., properties storing the location and background color; methods for hiding, showing and painting the widget. We can also query `window` for the interfaces it implements:

```
interface.GObject(window)

[1] "GtkBuildable"      "AtkImplementorIface"
```

When the underlying GTK+ object is destroyed, i.e., deleted from memory, the class of the proxy object is set to `<invalid>`, indicating that it can no longer be manipulated.

6.3 Methods

The next steps in our example are to create a “Hello World” button and to place the button in the window that we have already created. This depends on an understanding of how one programmatically manipulates widgets by invoking methods. Methods are functions that take an instance of their class as the first argument and instruct the widget to perform an action.

Although class information is stored in the style of S3, RGtk2 introduces its own mechanism for method dispatch. The call `obj$method(...)` resolves to a function call `f(obj, ...)`. The function is found by looking for any function that matches the pattern *classNameMethodName*, the concatenation of one of the names from `class(obj)` or `interface(obj)` with the method name. The search begins with the interfaces and proceeds through each character vector in order.

For instance, if `win` is a `gtkWindow` instance, then to resolve the call `win$add(widget)` RGtk2 considers `gtkBuildableAdd`, `atkImplementorIfaceAdd`, `gtkWindowAdd`, `gtkBinAdd` and finally finds `gtkContainerAdd`, which is called as `gtkContainerAdd(win, widget)`. The `$` method for RGtk2 objects does the work.

We take advantage of this convenience when we add the “Hello World” button to our window and set its size:

```
button <- gtkButton("Hello World")
window$add(button)
window$setDefaultSize(200, 200)
```

The above code calls the `gtkContainerAdd` and `gtkWindowSetDefaultSize` functions with less typing and less demands on the memory of the user.

Understanding this mechanism allows us to add to the RGtk2 API. For instance, we can add to the button API with

```
gtkButtonSayHello <- function(obj, target)
  obj$setLabel(paste("Hello", target))
button$sayHello("John")
button$getLabel()
```

```
[1] "Hello John"
```

Some common methods are inherited by most widgets, as they are defined in the base `gtkWidget` class. These include the methods `show` to specify that the widget should be drawn; `hide` to hide the widget until specified; `destroy` to destroy a widget and clear up any references to it; `getParent` to find the parent container of the widget; `modifyBg` to modify the background color of a widget; and `modifyFg` to modify the foreground color.

6.4 Properties

The GTK+ API uses properties to store object state. Properties are similar to R attributes and even more so to S4 slots. They are inherited, typed, self-describing and encapsulated, so that an object can intercept access to the underlying data. A list of properties that a widget has is returned by its `getPropInfo` method. RGtk2 provides the R generic names as a familiar alternative for this method. Auto-completion of property names is gained as a side effect. For the button just defined, we can see the first eight properties listed with:

```
head(names(button), n=8)           # or b$getPropInfo()

[1] "user-data"      "name"           "parent"         "width-request"
[5] "height-request" "visible"        "sensitive"      "app-paintable"
```

Some common properties are: `parent`, to store the parent widget (if any); `user-data`, which allows one to store arbitrary data with the widget; and `sensitive`, to control whether a widget can receive user events.

There are a few different ways to access these properties. GTK+ provides the functions `gObjectGet` and `gObjectSet` to get and set properties of a widget. The set function treats the arguments names as the property names, and setting multiple properties at once is supported. Here we add an icon to the top-left corner of our window and set the title:

```
image <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
window$set(icon = image, title = "Hello World 1.0")
```

Additionally, most user-accessible properties have specific get and set methods defined for them. For example, to set the title of the window, we could have used the `setTitle` method and verified the change with `getTitle`.

```
window$setTitle("Hello World 1.0")
window$getTitle()
```

```
[1] "Hello World 1.0"
```

RGtk2 provides the convenient and familiar `[` and `[<-` methods to get and access the properties. In our example, we might check the window to ensure that it is not yet visible:

```
window["visible"]
```

```
[1] FALSE
```

Finally, we can make our window visible by setting the “visible” property, although calling `gtkWidgetShow` is more conventional:

```
window["visible"] <- TRUE
window$show() # same effect
```

For ease of referencing the appropriate help pages, we tend to use the full method name in the examples, although at times the move R-like vector notation will be used for commonly accessed properties.

6.5 Events and signals

In RGtk2, a user action, such as a mouse click, key press, or drag and drop motion, triggers the widget to emit a corresponding signal. A GUI can be made interactive by specifying a callback function to be invoked upon the emission of a particular signal.

The signals provided by a class or interface are returned by the function `gTypeGetSignals`. For example

```
names(gTypeGetSignals("GtkButton"))
```

shows the “clicked” signal in addition to others. Note that this only lists the signals provided directly by the `GtkButton`. To list all inherited signals, we need to loop over the hierarchy, but it is not common to do this in practice, as the documentation includes information on the signals.

The `gSignalConnect` (or `gSignalConnect`) function is used to add a callback to a widget’s signal. Its signature is

```
args(gSignalConnect)
```

```
function (obj, signal, f, data = NULL, after = FALSE, user.data.first = FALSE)
```

The basic usage is to call `gSignalConnect` to connect a callback function `f` to the signal named `signal` belonging to the object `obj`. The function returns an identifier for managing the connection. This is not usually necessary but will be discussed later.

We demonstrate this usage by adding a callback to our “Hello World” example, so that “Hello World” is printed to the console when the button is clicked:

```
gSignalConnect(button, "clicked",  
               function(widget) print("Hello world!"))
```

```
clicked  
      12  
attr(,"class")  
[1] "CallbackID"
```

The `data` argument allows arbitrary data to be passed to the callback. The `user.data.first` argument specifies if this data argument should be the first argument to the callback or (the default) the last.

The `after` argument is a logical indicating if the callback should be called after the default handlers (see `?gSignalConnect`).

The signature for the callback varies for each signal. Unless `user.data.first` is `TRUE`, the first argument is the widget. Other arguments are possible depending on the signal type. For window events, the second argument is a `GdkEvent` type, which can carry with it extra information about the event that occurred. The GTK+ API lists the signature of each signal.

It is important to note that the widget, and possibly other arguments, are references, so their manipulation has side effects outside of the callback. This is obviously a critical feature, but it is one that may be surprising to the R user.

```
w <- gtkWindow(); w['title'] <- "test signals"
x <- 1;
b <- gtkButton("click me"); w$add(b)
ID <- gSignalConnect(b, signal = "clicked", f = function(widget) {
  widget$setData("x", 2)
  x <- 2
  return(TRUE)
})
```

Then after clicking, we would have

```
cat(x, b$getData("x"), "\n") # 1 and 2
```

```
1 2
```

Callbacks for signals emitted by window manager events are expected to return a logical value. Failure to do so can cause errors to be raised. For most other callbacks the return value is ignored, so it is safe to always return a logical value. For window events, a return value of `TRUE` indicates that no further callbacks should be called, whereas `FALSE` indicates that the next callback should be called. So in the following example, only the first two callbacks are executed when the user presses on the button.

```
b <- gtkButton("click")
w <- gtkWindow()
w$add(b)
id1 <- gSignalConnect(b, "button-press-event",
  function(b, event, data) {
    print("hi"); return(FALSE)
  })
id2 <- gSignalConnect(b, "button-press-event",
  function(b, event, data) {
    print("and"); return(TRUE)
  })
id3 <- gSignalConnect(b, "button-press-event",
  function(b, event, data) {
    print("bye"); return(TRUE)
  })
```

Multiple callbacks can be assigned to each signal. They will be processed in the order they were bound to the signal. The `gSignalConnect` function returns an ID that can be used to disconnect a callback if desired using `gSignalHandlerDisconnect` or temporarily blocked using `gSignalHandlerBlock` and `gSignalHandlerUnblock`. The man page for `gSignalConnect` gives the details on this, and much more.

6.6 Enumerated types and flags

At the beginning of our example, we constructed the window thusly:

```
window <- gtkWindow("toplevel", show = FALSE)
```

The first parameter indicates the window type. The set of possible window types is specified by what in C is known as an *enumeration*. A value from an enumeration can be thought of as a length one factor in R. The possible values defined by the enumeration are analogous to the factor levels. Since enumerations are foreign to R, RGtk2 accepts string representations of enumeration values, like "toplevel".

For every GTK+ enumeration, RGtk2 provides an R vector that maps the nicknames to the underlying numeric values. In the above case, the vector is named `GtkWindowType`.

```
GtkWindowType
```

```
toplevel  popup
      0      1
attr(,"class")
[1] "enums"
```

The names of the vector indicate the allowed nickname for each value of the enumeration. It is rarely necessary to explicitly use the enumeration vectors; specifying the nickname will work in most cases, including all method invocations, and is preferable as it is easier for human readers to comprehend.

Flags are an extension of enumerations, where the value of each member is a unique power of two, so that the values can be combined unambiguously. An example of a flag enumeration is `GtkWidgetFlags`.

```
GtkWidgetFlags
```

	toplevel	no-window	realized	mapped
	16	32	64	
128	visible	sensitive	parent-sensitive	can-focus
	256	512	1024	2048
	has-focus	can-default	has-default	has-grab
	4096	8192	16384	32768
	rc-style	composite-child	no-reparent	app-paintable

```

      16384      131072      262144      524288
receives-default double-buffered      no-show-all
      1048576      2097152      4194304
attr(,"class")
[1] "flags"

```

GtkWidgetFlags represents the possible flags that can be set on a widget. We can retrieve the flags currently set on our window:

```

window$flags()

[1] 2164688
attr(,"class")
[1] "GtkWidgetFlags" "flag"

```

Flag values can be combined using | the bitwise OR. The & function, the bitwise AND, allows one to check whether a value belongs to a combination. For example, we could check whether our window is top-level:

```

(window$flags() & GtkWidgetFlags["toplevel"]) > 0

[1] TRUE

```

6.7 The event loop

RGtk2 integrates the GTK+ eventloop with the R event loop. A separate thread continuously iterates the GTK+ event loop, in synchronization with the main R thread. Thus, if the R thread is busy, the GTK+ event loop will not be iterated. During a long calculation, the GUI can seem unresponsive. To avoid this, the following construct should be inserted into the long running algorithm in order to ensure that GTK+ events are periodically processed.

```

while(gtkEventsPending())
  gtkMainIteration()

```


RGtk2: Basic Components

This section covers some of the basic widgets and containers of GTK+. We begin with a discussion of top level containers and box containers. Then we describe many of the basic controls, and conclude with the mention of a few special-case containers.

7.1 Top-level windows

As we saw in our “Hello World” example, top-level windows are constructed by the `gtkWindow` constructor. This function has arguments `type` to specify the type of window to create. The default is a top-level window, which we will always use, as the alternative is for “popups” which are meant for internal use, e.g., for implementing menus. The second argument is `show`, which by default is `TRUE`, indicating that the window should be shown. If set to `FALSE`, the window, like other widgets, can later be shown by calling its `show` method. The `showAll` method will also show any child components. These can be reversed with `hide` and `hideAll`.

As with all objects, windows have several properties. The window title is stored in the `title` property. As usual, this property can be accessed via the “get” and “set” methods `getTitle` and `setTitle`, or using the `[]` function. To illustrate, the following sets up a new window with a title.

```
w <- gtkWindow(show=FALSE)           # use default type
w$setTitle("Window title")           # set window title
w['title']                           # or w$getTitle()

[1] "Window title"

w$setDefaultSize(250,300)             # 250 wide , 300 high
w$show()                             # show window
```

Window size The initial size of the window can be set with the `setDefaultSize` method, as shown, which takes a `width` and `height` argument specified in

pixels. This specification allows the window to be resized, but must be made before the window is drawn, as the window then falls under control of the window manager. The `setSizeRequest` method will request a minimum size, which the window manager will usually honor, as long as a maximum bound is not violated. To fix the size of a window, the `resizable` property may be set to `FALSE`.

Adding a child component to a window A window is a container. `GtkWindow` inherits from `GtkBin`, which can contain only a single child. As before, this child is added by the `add` method. To display multiple widgets in a window, one simply needs to add a non-`GtkBin` container as the child widget.

We illustrate the basics by adding a simple label to a window.

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
```

Destroying windows A window is normally closed by the window manager. Most often, this occurs in response to the user clicking on a close button in a title bar. It is also possible to close a window programatically by calling its `destroy` method. When the user clicks on the close button, the window manager requests that the window be deleted, and the `delete-event` signal is emitted. The contract of deletion is that the window should no longer be visible on the screen. It is not necessary for the actual window object to be removed from memory, although this is the default behavior. Calling the `hideOnDelete` method configures the window to hide but not destroy itself. As with any event, the default handler is overridden if a callback connected to `delete-event` returns `TRUE`. This can be useful for confirming the intention of the user before closing the window.

Transient windows New windows may be standalone top-level windows, or may be associated with some other window. For example, a dialog is usually associated with the primary document window. The `setTransientFor` method can be used to specify the window with which a transient (dialog) window is associated. This hints the window manager that the transient window should be kept on top of its parent. The position relative to the parent window can be specified with `setPosition`, which takes a value from the `GtkWindowPosition` enumeration. Optionally, a dialog can be set to be destroyed with its parent. For example:

```
## create a window and a dialog window
w <- gtkWindow(show=FALSE); w$setTitle("Top level window")
d <- gtkWindow(show=FALSE); d$setTitle("dialog window")
d$setTransientFor(w)
d$setPosition(GtkWindowPosition["center-on-parent"])
```

```
d$setDestroyWithParent(TRUE)
w$show()
d$show()
```

The above code produces a non-modal dialog window from scratch. Due to its transient nature, it can hide parts of the top-level window, but, unlike a modal dialog, it does not prevent that window from receiving events. GTK+ provides a number of convenient high-level dialogs, discussed later, that support modal operation.

7.2 Layout containers

Once a top-level window is constructed, it remains to fill the window with the controls that will constitute our GUI. As these controls are graphical, they must occupy a specific region on the screen. The region could be specified explicitly, as a rectangle. However, as a user interface, a GUI is dynamic and interactive. The size constraints of widgets will change, and the window will be resized. The programmer cannot afford to explicitly manage a dynamic layout. Thus, GTK+ implements automatic layout in the form of container widgets.

Basics

The method `getChildren` will return the children of a container as a list. Since in this case the list will be at most length one, the `getChild` method may be more convenient, as it directly returns the only child, if any. For instance, to retrieve the label text one could do:

```
w$getChild()['label']           # return label property of child
NULL
```

The `[]` method accesses the child containers by number, as a convenient wrapper around the `getChildren` method.

In GTK+, the widget hierarchy is built when children are added to a parent container. In our example, the window is the immediate parent of the label. The `getParent` method for GTK+ widgets will return the parent container of a widget.

Every container supports removing a child with the `remove` method. The child can later be re-added using `packStart`. For instance

```
b <- g[[3]]
g$remove(b)           # removed
g$packStart(b, expand=TRUE, fill=TRUE)
```

To remove a widget from the screen but not its container, use the `hide` method on the widget. This can be reversed with the `show` method. The `reparent` method is a convenience for moving a widget between containers.

Widget size negotiation

We have already seen perhaps the simplest automatic layout container, `GtkWindow`, which fills all of its space with its child. While simple, there is a considerable amount of logic for calculating the size of the widget on the screen. The child will first inform the parent of its desired natural size. For example, a label might ask for the dimensions necessary to display all of its text. The container then decides whether to allocate the requested size or to allocate more or less than the requested amount. The child consumes the allocated space. Consider the previous example of adding a label to a window:

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
```

The window is shown before the label is added, and the default size is likely much larger than the space the label needs to display “Hello world”. However, as the window size is now controlled by the window manager, `GtkWindow` will not adjust its size. Thus, the label is allocated more space than it requires.

```
l$getAllocation()
```

```
$x
[1] -1

$y
[1] -1

$width
[1] 1

$height
[1] 1

attr(,"class")
[1] "GtkAllocation"
```

If, however, we avoid showing the window until the label is added, the window will size itself so that the label has its natural size:

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
w$show()
l$getAllocation()
```

```
$x
[1] 0
```

```
$y
[1] 0

$width
[1] 79

$height
[1] 18

attr(,"class")
[1] "GtkAllocation"
```

One might notice that it is not possible to decrease the size of the window further. This is due to `GtkLabel` asserting a minimum size request that is sufficient to display its text. The `setSizeRequest` sets a user-level minimum size request for any widget. It is obvious from the method name, however, that this is still strictly a request. It may not be satisfied, for example, if the maximum window size constraint of the window manager is violated. More importantly, setting a minimum size request is generally discouraged, as it decreases the flexibility of the layout.

Any non-trivial GUI will require a window containing multiple widgets. Let us consider the case where the child of the window is itself a container, with multiple children. Essentially the same negotiation process occurs between the container and its children (the grandchildren of the window). The container calculates its size request based on the requests of its children and communicates it to the window. The size allocated to the container is then distributed to the children according to its layout algorithm. This process is the same for every level in the container hierarchy.

Box containers

The most commonly used multichild container in GTK+ is the box, `GtkBox`, which packs its children as if they were in a box. Instances of `GtkBox` are constructed by `gtkHBox` or `gtkVBox`. These produce horizontal or vertical “boxes”, respectively. Each child widget is allocated a cell in the box. The cells are arranged in a single column (`GtkVBox`) or row (`GtkHBox`). This one dimensional stacking is usually all that a layout requires. The child widgets can be containers themselves, allowing for very flexible layouts. For special cases where some widgets need to span multiple rows or columns, GTK+ provides the `GtkTable` class, which is discussed later. Many of the principles we discuss in this section also apply to `GtkTable`.

Here we will explain and demonstrate the use of `GtkHBox`, the general horizontal box layout container. `GtkVBox` can be used exactly the same way; only the direction of stacking is different. Figure 7.1 illustrates a sampling of

the possible layouts that are possible with a `GtkHBox`.

Figure 7.1: A screenshot demonstrating the effect of packing two buttons into `GtkHBox` instances using the `packStart` method with different combinations of the `expand` and `fill` settings. The effect of the homogeneous spacing setting on the `GtkHBox` is also shown.

The code for some of these layouts is presented here. We begin by creating a `GtkHBox` widget. We pass `TRUE` for the first parameter, `homogeneous`. This means that the horizontal allocation of the box will be evenly distributed between the children. The second parameter directs the box to leave 5 pixels of space between each child. The following code constructs the `GtkHBox`:

```
box <- gtkHBox(TRUE, 5)
```

The equal distribution of available space is strictly enforced; the minimum size requirement of a homogeneous box is set such that the box always satisfies this assertion, as well as the minimum size requirements of its children.

The `packStart` and `packEnd` methods pack a widget into a box with left and right justification (top and bottom for a `GtkVBox`), respectively. For this explanation, we restrict ourselves to `packStart`, since `packEnd` works the same except for the justification. Below, we pack two buttons, `button_a` and `button_b` using left justification:

```
button_a <- gtkButton("Button A")
button_b <- gtkButton("Button B")
box$packStart(button_a, fill = FALSE)
box$packStart(button_b, fill = FALSE)
```

First, `button_a` is packed against the left side of the box, and then we pack `button_b` against the right side of `button_a`. This results in the first row in Figure 7.1. The space distribution is homogeneous, but making the space available to a child does not mean that the child will fill it. That depends on the minimum size requirement of the child, as well as the value of the `fill` parameter passed to `packStart`. In this case, `fill` is `FALSE`, so the extra space is not filled. When a widget is packed with the `fill` parameter set to `TRUE`, the widget is sized to consume the available space. This results in rows 2 and 3 in Figure 7.1.

In many cases, it is desirable to give children unequal amounts of available space, as in rows 4–9 in Figure 7.1. To create an inhomogeneously spaced `GtkHBox`, we pass `FALSE` as the first argument to the constructor, as in the following code:

```
box <- gtkHBox(FALSE, 5)
```

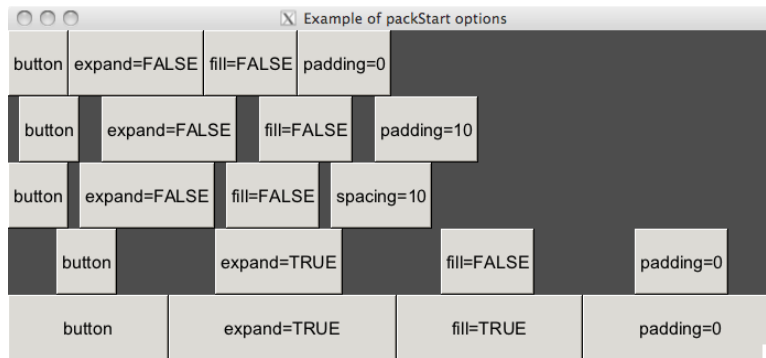


Figure 7.2: Examples of packing widgets into a box container. The top row shows no padding, whereas the 2nd and 3rd illustrate the difference between padding (an amount around each child) and spacing (an amount between each child). The last two rows show the effect of `fill` when `expand=TRUE`. This illustration follows one in original GTK+ tutorial.

An inhomogeneous layout is freed of the restriction that all widgets must be given the same amount of available space; it only needs to ensure that each child has enough space to meet its minimum size requirement. After satisfying this constraint, a box is often left with extra space. The programmer may control the distribution of this extra space through the `expand` parameter to `packStart`. When a widget is packed with `expand` set to `TRUE`, we will call the widget an *expanding* widget. All expanding widgets in a box are given an equal portion of the entirety of the extra space. If no widgets in a box are expanding, as in row 5 of Figure 7.1, the extra space is left undistributed.

It is common to mix expanding and non-expanding widgets in the same box. An example is given below, where `button_a` is expanding, while `button_b` is not:

```
box$packStart(button_a, expand = TRUE, fill = FALSE)
box$packStart(button_b, expand = FALSE, fill = FALSE)
```

The result is shown in row 6 of Figure 7.1. The figure contains several other permutations of the homogeneous, `expand` and `fill` settings.

There are several ways to add space around widgets in a box container. The `spacing` argument for the constructors specifies the amount of space, in pixels, between the cells. This defaults to zero. The `pack` methods have a `padding` argument, also defaulting to zero, for specifying the padding in pixels on either side of the child. It is important to note the difference: `spacing` is between children and the same for every boundary, while the `padding` is specific to a particular child and occurs on either side, even on the ends. The spacing between widgets is the sum of the `spacing` value and the two padding values when the children are added. Example 7.3 provides an ex-

ample and Figure 7.2 an illustration.

The `reorderChild` method can be used to reorder the child widgets. The new position of the child is specified using 0-based indexing. This code will move the last child to the second position.

```
b3 <- g[[3]]  
g$reorderChild(b3, 2 - 1)           # second is 2 - 1
```

Alignment

We began this section with a simple example of a window containing a label:

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")  
l <- gtkLabel("Hello world")  
w$add(l)
```

The window allocates all of its space to the label, despite the actual text consuming a much smaller region. The size of the text is fixed, according to the font size, so it could not be expanded. Thus, the label decided to center the text within itself (and thus the window). A similar problem is faced by widgets displaying images. The image cannot be expanded without distortion. Widgets that display objects of fixed size inherit from `GtkMisc`, which provides methods and properties for tweaking how the object is aligned within the space of the widget. For example, the `xalign` and `yalign` properties specify how the text is aligned in our label and take values between 0 and 1, with 0 being left and top. Their defaults are 0.5, for centered alignment. We modify them below to make our label left justified:

```
l["xalign"] <- 0
```

Unlike a block of text or an image, a widget usually does not have a fixed size. However, the user may wish to tweak how a widget fills the space allocated by its container. GTK+ provides the `GtkAlignment` container for this purpose. For example, rather than adjust the justification of the label text, we could have instructed the layout not to expand but to position itself against the left side of the window:

```
w <- gtkWindow(); w$setTitle("Hello world")  
a <- gtkAlignment()  
a$set(xalign = 0, yalign = 0.5, xscale = 0, yscale = 1)  
w$add(a)  
l <- gtkLabel("Hello world")  
a$add(l)
```

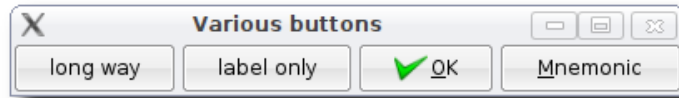



Figure 7.3: Various buttons

7.3 Buttons

The button is the very essence of a GUI. It communicates its purpose to the user and executes a command in response to a simple click or key press. In GTK+, a basic button is usually constructed using `gtkButton`, as the following example demonstrates.

Example 7.1: Button constructors

```
w <- gtkWindow(show=FALSE)
w$setTitle("Various buttons")
w$setDefaultSize(400, 25)
g <- gtkHBox(homogeneous=FALSE, spacing=5)
w$add(g)
b <- gtkButtonNew()
b$setLabel("long way")
g$packStart(b)
g$packStart(gtkButton(label="label only") )
g$packStart(gtkButton(stock.id="gtk-ok") )
g$packStart(gtkButtonNewWithMnemonic("_Mnemonic") ) # Alt-m to "click"
w$show()
```

A `GtkButton` is simply a clickable region on the screen that is decorated to appear as a button. `GtkButton` is a subclass of `GtkBin`, so it will accept any widget as an indicator of its purpose. By far the most common button decoration is a label. The first argument of `gtkButton`, `label`, accepts the text for an automatically created `GtkLabel`. We have seen this usage in our “Hello World” example and others.

The alternative `stock.id` argument will use decorations associated with the stock identifier. For example, “`gtk-ok`” would produce a button with a theme-dependent image (such as a checkmark) and the “Ok” label, with the appropriate mnemonic and translated into the current language. The available stock identifiers are listed by `gtkStockListIds`. See `help(“stock-items”)` for more information.

The final button created in the example uses `gtkButtonNewWithMnemonic` to create a button with a mnemonic. Mnemonics are specified by prefixing the character with an underscore.

The method `setRelief` changes the relief style of the button. For example, the relief can be disabled so that the button is drawn like a label.

Signals The `clicked` signal is emitted when the button is clicked on with the mouse or when the button has focus and the enter key is pressed. A callback can listen for this event to perform a command when the button is clicked.

Example 7.2: Callback example for `gtkButton`

```
w <- gtkWindow(); b <- gtkButton("click me");
w$add(b)
ID <- gSignalConnect(b,"button-press-event", # just mouse click
                    f = function(w,e,data) {
                        print(e$getButton()) # which button
                        return(FALSE)        # propagate
                    })
ID <- gSignalConnect(b,"clicked",           # click or keyboard
                    f = function(w,...) {
                        print("clicked")
                    })
```

As buttons are intended to call an action immediately after being clicked, it is customary to make them insensitive to user input when the action is not possible. The `setSensitive` method can adjust this for the button, as with other widgets.

Windows often have a default action. For example, if a window contains a form, the default action often submits the form. If the action a button is to initiate is the default action for the window it can be set so that it is activated when the user presses enter while the parent window has the focus. To implement this, the property `can-default` must be `TRUE` and the widget method `grabDefault` must be called. (This is not specific to buttons, but any widget that can be activatable.)

If the action that a button initiates is to be represented elsewhere in the GUI, say a menu bar, then a `GtkAction` object may be appropriate. Action objects are covered in Section 9.4.

Example 7.3: Spacing between buttons

This example shows how to pack buttons into a box so that the spacing between the similar buttons is 12 pixels, but between potentially dangerous buttons is 24 pixels, as per the Mac human interface guidelines. GTK+ provides the constructor `gtkHButtonBox` for holding buttons, which provides a means to apply consistent styles, but the default styles do not allow such spacing as desired. (Had all we wanted was to right align the buttons, then that style is certainly supported.) As such, we will illustrate how this can

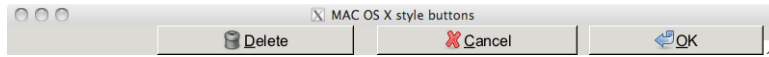


Figure 7.4: Example using stock buttons with extra spacing added between the delete and cancel buttons.

be done through a combination of spacing arguments. We assume that our parent container, `g`, is a horizontal box container.

We include standard buttons, so use the stock names and icons.

```
cancel <- gtkButton(stock.id="gtk-cancel")
ok <- gtkButton(stock.id="gtk-ok")
delete <- gtkButton(stock.id="gtk-delete")
```

We will right align our buttons, so use the parent container's `PackEnd` method. The `ok` button has no padding, the 12-pixel gap between it and the `cancel` button is ensured by the `padding` argument when the `cancel` button is added. Treating the `delete` button as potentially irreversible, we aim to have 24 pixels of separation between it and the `cancel` button. This is given by adding 12 pixels of padding when this button is packed in, giving 24 in total. The blank label is there to fill out space if the parent container expands.

```
g$packEnd(ok, padding=0)
g$packEnd(cancel, padding=12)
g$packEnd(delete, padding=12)
g$packEnd(gtkLabel(""), expand=TRUE, fill=TRUE)
```

We make `ok` the default button, so have it grab the focus and add a simple callback when the button is either clicked or the enter key is pressed when the button has the focus.

```
ok$grabFocus()
QT <- gSignalConnect(ok, "clicked", function(...) print("ok"))
```

7.4 Static Text and Images

Labels

The primary purpose of a label is to communicate the role of another widget, as we showed for the button. Labels are created by the `gtkLabel` constructor. Its main argument is `str` to specify the button text, stored in the `label` property. This text can be set with either `setLabel` or `setText` and retrieved with either `getLabel` or `getText`. The difference being the former respects formatting marks.

Example 7.4: Label formatting

As all text in a GTK+ GUI is ultimately displayed by `GtkLabel`, there are

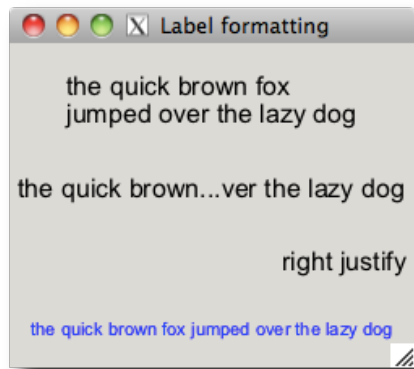


Figure 7.5: Various formatting for a label: wrapping, alignment, ellipsizing, PANGO markup

many formatting options available. This example demonstrates a sample of these (Figure ??)>

```
w <- gtkWindow(); w$setTitle("Label formatting")
w$setSizeRequest(250,100) # narrow
g <- gtkVBox(spacing=2); g$setBorderWidth(5); w$add(g)
string <- "the quick brown fox jumped over the lazy dog"
## wrap by setting number of characters
basicLabel <- gtkLabel(string)
basicLabel$setLineWrap(TRUE)
basicLabel$setWidthChars(35) # specify number of characters
## Set ellipsis to shorten long text
ellipsized <- gtkLabel(string)
ellipsized$setEllipsize(PangoEllipsizeMode["middle"])
## Right justify text lines
## use xalign property for aligning entire block
rightJustified <- gtkLabel("right justify");
rightJustified$setJustify(GtkJustification["right"])
rightJustified['xalign'] <- 1
## PANGO markup
pangoLabel <- gtkLabel()
pangoLabel$setMarkup(paste("<span foreground='blue' size='x-small'>",
                           string, "</span>"))
sapply(list(basicLabel, ellipsized, rightJustified, pangoLabel),
       g$packStart, expand = TRUE, fill = TRUE)
```

```
[[1]]
NULL
```

```
[[2]]
```

```
NULL
```

```
[[3]]
NULL
```

```
[[4]]
NULL
```

```
w$showAll()
```

Many of the text formatting options are demonstrated in Example 7.4. Line wrapping is enabled with `setLineWrap`. Labels also support explicit line breaks, specified with `"\n."` The `setWidthChar` method is a convenience for instructing the label to request enough space to show a specified number of characters in a line. When space is at a premium, long labels can be ellipsized, i.e., have their text truncated and appended with an ellipsis, "...". By default this is turned off; to enable, call `setEllipsize`. The property `justify`, with values taken from `GtkJustification`, controls the alignment of multiple lines within a label. To align the entire block of text within the space allocated to the label, modify the `xalign` property, as described in Section 7.2.

GTK+ allows markup of text elements using the Pango text attribute markup language, an XML-based format that resembles basic HTML. The method `setMarkup` accepts text in the format. Text is marked using tags to indicate the style. Some convenient tags are `` for bold, `<i>` for italics, `<u>` for underline, and `<tt>` for monospace text. More complicated markup involves the `` tag markup, such as `some text`. The text can may need to be escaped first, so that designated entities replace reserved characters.

Although mostly meant for static text display, `GtkLabel` has some interactive features. If the `selectable` property is set to `TRUE`, the text can be selected and copied into the clipboard. Labels can hold mnemonics for other widgets; this is useful for navigating forms. The mnemonic is specified at construction time with `gtkLabelNewWithMnemonic`. The `setMnemonicWidget` method identifies the widget to which the mnemonic refers.

Signals Unlike buttons, labels do not emit any specific signals, as they are intended to hold static text. Although a label is a `GtkWidget`, it does not receive any system events. A work-around is to place the label within an instance of `GtkEventBox`. This creates a non-visible parent window for the label that listens to the windowing system. Example ?? will illustrate the use of an event box. Alternatively, if a clickable label is desired, one could use an instance of `gtkButton` with its `relief` property assigned to `GtkReliefStyle['none']`.

Statusbars

In GTK+, a statusbar is constructed through the `gtkStatusbar` function. Statusbars must be placed at the bottom of a top-level window by the programmer. In GTK+, a statusbar keeps various stacks of messages for display. One adds a message to display for given stack through the `Push` method by specifying first an integer value for `context.id` and a message. To pop the top message on a stack and display the next, the method `Pop` method is available.

Images

It is said that a picture can be worth a thousand words, and images are often a more space efficient alternative to labels. `GtkImage` is the widget that displays images. The constructor `gtkImage` supports creating images from various in-memory image representations, files, and other sources. We only discuss loading an image from a file. Images can be loaded after construction, as well. For example, the `setFromFile` method loads an image from a file.

The image widget, like the label widget, does not have a parent `GdkWindow`, which means it does not receive window events. As with the label widget, the image widget can be placed inside a `gtkEventBox` container if one wishes to connect to such events.

Example 7.5: Using a pixmap to present graphs

This example shows how to use a `GtkImage` object to embed a graphic within RGtk2, using the `cairoDevice` package. The basic idea is to draw onto an off-screen pixmap using `cairoDevice` and then to construct a `GtkImage` from the pixmap.

We begin by creating a window of a certain size.

```
w <- gtkWindow(show=FALSE); w$setTitle("Graphic window");
w$setSizeRequest(400,400)
g <- gtkHBox(); w$add(g)
w$showAll()
```

The size of the image is retrieved from the size allocated to the box `g`. This allows the window to be resized prior to drawing the graphic. Unlike an interactive device, after drawing, this graphic does not resize itself when the window resizes.

```
theSize <- g$getAllocation()
width <- theSize$width; height <- theSize$height
```

Now we create a `GdkPixmap` of the correct dimensions and initialize an R graphics device that targets the pixmap. We then draw a simple histogram using base R graphics.

```
require(cairoDevice)
```

```

pixmap <- gdkPixmap(drawable = NULL, width = width, height = height,
                    depth = 24)
asCairoDevice(pixmap)

```

```
[1] TRUE
```

```
hist(rnorm(100))
```

The final step is to create the `GtkImage` widget to display the pixmap:

```

image <- gtkImage(pixmap = pixmap)
g$packStart(image, expand=TRUE, fill = TRUE)

```

Stock icons

In GTK+, standard icons, like the one on the “OK” button, can be customized by themes. This is implemented by a database that maps a *stock* identifier to an icon image. The stock identifier corresponds to a commonly performed type of action, such as the “OK” response or the “Save” operation. There is no hard-coded set of stock identifiers, however GTK+ provides a default set for the most common operations. These identifiers are all prefixed with “gtk-”. Users may register new types of stock icons.

As mentioned previously, the full list of stock icons are returned in a list by `gtkStockListIds`. The first 4 are:

```
head(unlist(gtkStockListIds()), n=4)
```

```
[1] "gtk-zoom-out" "gtk-zoom-in" "gtk-zoom-fit" "gtk-zoom-100"
```

The use of stock identifiers over specific images is encouraged, as it allows an application to be customized through themes. The `gtkButton` and `gtkImage` constructors accept a stock identifier passed as `stock.id` argument, and the icons in toolbars and menus are most conveniently specified by stock identifier.

Example 7.6: Adding to the stock icons

This example shows, without much explanation the steps to add images to the list of stock icons. To generate some sample icons, we use those provided by objects in the `ggplot2` package.

First we create the icons using the fact that the objects have a function `icon` to draw an image.

```

require(ggplot2)
iconNames <- c("GeomBar", "GeomHistogram") # 2 of many ggplot functions
icon.size <- 16
pixbufs <- sapply(iconNames, function(name) {
  pixmap <- gdkPixmap(drawable = NULL, width = icon.size, height = icon.size,
                      depth = 24)

```

7. RGtk2: BASIC COMPONENTS

```
asCairoDevice(pixmap)
val <- try(get(name))
grid.newpage()
try(grid.draw(val$icon()), silent=TRUE)
dev.off()
gdkPixbufGetFromDrawable(NULL, pixmap, NULL, 0, 0, 0, 0, -1 -1)
})
```

The following function works through the steps to add a new icon. The basic ideas are sketched out in the API for `GtkIconSet`.

```
addToStockIcons <- function(pixbufs, stock.prefix="new") {
  iconfactory <- gtkIconFactory()

  items <- lapply(names(pixbufs), function(iconName) {
    ## each icon has its own icon set, which is registered with icon factory
    iconset <- gtkIconSetNewFromPixbuf(pixbufs[[iconName]])
    stockName <- paste(stock.prefix, "-", iconName, sep="")
    iconfactory$add(stockName, iconset)

    ## create stock item for icon
    as.GtkStockItem(list(stock_id = stockName, label = iconName))
  })
  ## register our factory of icons
  iconfactory$addDefault()
  ## officially register the stock items
  gtkStockAdd(items)
}
```

We call this function and then check that the values are added:

```
addToStockIcons(pixbufs)
nms <- gtkStockListIds()
unlist(nms[grepl("^new", nms)])
```

Example 7.7: An alert panel

This example puts together images, buttons, labels and box containers to create an alert panel, or information bar. This is an area that seems to drop down from the menu bar to give users feedback about an action that is less disruptive than a modal dialog. A similar widget is used in the Firefox browser with its popup blocker. Although, as of version 2.18, a similar feature is available in GTK+ through the `GtkInfoBar` widget, this example is given, as it shows how several useful things in GTK+ can be combined to customize the user experience.

This constructor for the widget specifies some properties and returns an environment to store these properties, as our function calls will need to update these properties and have be persistent.

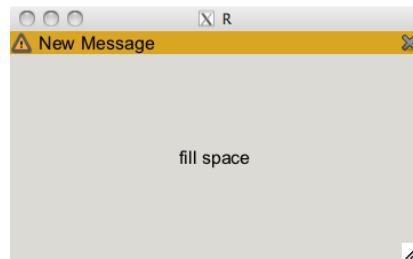


Figure 7.6: The alert panel showing a message.

```
newAlertPanel <- function(wrap=35,
                          icon="gtk-dialog-warning",
                          message="",
                          panel.color="goldenrod",
                          evb=NULL,
                          image=NULL,
                          label=NULL # info
                        ) {
  x <- c("wrap", "icon", "message", "panel.color", "evb", "image", "label")
  e <- new.env()
  sapply(x, function(i) assign(i, envir=e, get(i)))
  return(e)
}
```

An alert panel needs just a few methods: one to create the widget, one to show the widget and one to hide the widget. We create a function `getAlertPanelBlock` to return a component that can be added to a container. An event box is used so that we can color the background, as this isn't possible for a box container due to its lack of a gdk window. To this event box we add a box container that will hold an icon indicating this is an alert, a label for the message, and another icon to indicate to the user how to close the alert. Since we wish to receive mouse clicks on close icon, we place this inside another event box. To this, we bind a callback to the button-press-event signal.

```
getAlertPanelBlock <- function(obj) {

  obj$evb <- gtkEventBox(show=FALSE)
  obj$evb$ModifyBg(state="normal", color=obj$panel.color)

  g <- gtkHBox(homogeneous=FALSE, spacing=5)
  obj$evb$add(g)

  obj$image <- gtkImageNewFromStock(obj$icon, size="button")
  obj$image['yalign'] <- .5
  g$packStart(obj$image, expand=FALSE)
```

```
obj$label <- gtkLabel(obj$message)
obj$label['xalign'] <- 0; obj$label['yalign'] <- .5
obj$label$setLineWrap(TRUE)
obj$label$setWidthChars(obj$wrap)
g$packStart(obj$label, expand=TRUE, fill=TRUE)

xbutton <- gtkEventBox()
xbutton$modifyBg(state="normal", color=obj$panel.color)
xbutton$add(gtkImageNewFromStock("gtk-close", size="menu"))
g$packEnd(xbutton, expand=FALSE, fill=FALSE)
xbuttonCallback <- function(data, widget,...) {
  hideAlertPanel(data)
  return(FALSE)
}

## close on button press and event box click
sapply(list(xbutton, obj$evb), function(i) {
  gSignalConnect(i, "button-press-event",
                 f=xbuttonCallback,
                 data=obj, user.data.first=TRUE)
}))
return(obj$evb)
}
```

The `showAlertPanel` function updates the message and then calls the `Show` method of the event box.

```
showAlertPanel <- function(obj) {
  obj$label$setText(obj$message)
  obj$evb$show()
}
```

Our `hideAlertPanel` function simply calls the `hide` method the event box.

```
hideAlertPanel <- function(obj) obj$evb$hide()
```

To test it out, we create a simple GUI

```
w <- gtkWindow()
g <- gtkVBox(); w$add(g)
ap <- newAlertPanel()
g$packStart(getAlertPanelBlock(ap), expand=FALSE)
g$packStart(gtkLabel("fill space"), expand=TRUE, fill=TRUE)
ap$message <- "New Message" # add message
showAlertPanel(ap)
```

To improve this, one could also add a time to close the panel after some delay. The `gTimeoutAdd` function is used to specify a function to call periodically until the function returns `FALSE`.

7.5 Input Controls

Text entry

The widgets explained thus far are largely static. For example, GTK+ does not yet support editable labels. Text editing is handled by other widgets that are rendered in the familiar depressed box form. We will discuss complex multi-line text editing in Section 8.5. For entering a single line of text, the `GtkEntry` widget is appropriate. It is constructed by `gtkEntry`. An argument `max` specifies the maximum number of characters if positive, but this calls a deprecated function. Instead, call the method `setMaxLength` after construction.

The text property stores the text. This can be set with the method `setText` and retrieved with `getText`. Editing text programmatically relies on the `GtkEditable` interface, which `GtkEntry` implements. The method `insertText` inserts text. Its argument `new.text` contains the text and position specifies the position of the text to be added. The return value is a list with the component position indicating the position *after* the new text. The `deleteText` method deletes text. This takes two integers indicating the start and finish location of the text to delete.

Example 7.8: Insert and Delete text

The example will show how to add then delete text.

```
e <- gtkEntry()
e$setText("Where did that guy go?")
add.pos <- regexpr("guy", e['text']) - 1 # before "guy"
ret <- e$insertText("@$#! ", position = add.pos)
e$getText()                               # or e['text']
```

```
[1] "Where did that @$#! guy go?"
```

```
e$deleteText(start = add.pos, end= ret$position)
e$getText()
```

```
[1] "Where did that guy go?"
```

The `GtkEditable` interface supports three signals: `changed` when text is changed, `delete-text` for delete events, and `insert-text` for insert events. It is possible to prevent the insertion or deletion of text by connecting to the corresponding signal and stopping the signal propagation with `gSignalStopEmission`.

`GtkEntry` defines a number of its own signals, including the `activate` signal, which is emitted when the enter key is pressed.

Check button

Very often, the action performed by a button simply changes the value of a state variable in the application. GTK+ defines several types of buttons that explicitly manage and display the value of a state variable. The simplest type of state variable is binary (boolean) and is usually proxied by a `GtkCheckButton`.

A `GtkCheckButton` is constructed by `gtkCheckButton`. The optional argument `label` places a label next to the button. The alternative constructor `gtkCheckButtonNewWithMnemonic` gives the label a mnemonic.

As with any `GtkButton`, the `label` property stores the label. The state of the binary variable is represented by the `active` property. It can be set or retrieved with the methods `setActive` and `getActive`.

When the state is changed the `toggle` signal is emitted. The callback should check the `active` property to determine if the button has been enabled or disabled.

An alternative to `GtkCheckButton` is the lesser used `GtkToggleButton`, which is actually the parent class of `GtkCheckButton`. A toggle button is drawn as an ordinary button. It remains depressed while the state variable is `TRUE`, instead of relying on a checkbox to communicate the binary value.

Radio button group

GTK+ provides two button types for discrete state variables that accept more than two possible values: combo boxes, discussed in the next section, and radio buttons. The `gtkRadioButton` constructor creates an instance of `GtkRadioButton`, an extension of `GtkCheckButton`. Each radio button belongs to a group. There is no explicit group object; rather, the buttons are chained together as a linked list. By default, a newly constructed button is added to its own group. If `group` is a list of radio buttons, the newly created button is added to the group. The constructor returns a single radio button widget.

Like other types derived from `GtkToggleButton`, each radio button in the group has an `active` property. Only one button in the group can have `active` set to `TRUE` at a time. To determine which button is active, each button needs to be queried individually. Setting `active` to `TRUE` activates the corresponding button and ensures that the other buttons are disabled.

Example 7.9: Radio group construction

Creating a new radio button group with the basic `gtkRadioButton` constructor follows this pattern:

```
vals <- c("two.sided", "less", "greater")
radiogp <- list()                                # list for group
radiogp[[vals[1]]] <- gtkRadioButton(label=vals[1]) # group = NULL
for(i in vals[-1])
```

```
radiogp[[i]] <- gtkRadioButton(radiogp, label=i) # group is a list
```

Each button needs to be managed. Here we illustrate a simple GUI doing so.

```
w <- gtkWindow(); w$setTitle("Radio group example")
g <- gtkVBox(FALSE, 5); w$add(g)
sapply(radiogp, gtkBoxPackStart, object = g)
```

```
$two.sided
NULL
```

```
$less
NULL
```

```
$greater
NULL
```

We can set and query which button is active, as follows:

```
g[[3]]$setActive(TRUE)
sapply(radiogp, '[', "active")
```

```
two.sided    less    greater
FALSE        FALSE    TRUE
```

Here is how we might register a callback for the toggled signal.

```
sapply(radiogp, gSignalConnect, "toggled", # attach each to "toggled"
      f = function(w, data) {
        if(w$getActive()) # set before callback
          cat("clicked", w$getLabel(), "\n")
      })
```

```
two.sided.toggled    less.toggled    greater.toggled
135                  136              137
```

The `getGroup` method returns a list containing the radio buttons in the same group. However, it is in the reverse order of construction (newest first). This results from an internal optimization that prepends, rather than appends, the buttons to a linked list.

As a convenience, there are constructor functions ending with `FromWidget` that determine the group from a radio button belonging to the group. As we will see in our second example, this allows for a more natural `sapply` idiom that avoids the need to allocate a list and populate it in a `for` loop.

Example 7.10: Radio group using `getGroup`

In this example below, we illustrate two things: using the `gtkRadioButtonNewWithLabelFromWidget` function to add new buttons to the group and the `GetGroup` method to reference the buttons. The `rev` function is used to pack the widgets, to get them to display first to last.

```
radiogp <- gtkRadioButton(label=vals[1])
supply(vals[-1], gtkRadioButtonNewWithLabelFromWidget, group = radiogp)
```

```
$less
<pointer: 0x2cea238>
attr(,"interfaces")
[1] "GtkBuildable"          "AtkImplementorIface"
attr(,"class")
[1] "GtkRadioButton"      "GtkCheckButton"      "GtkToggleButton"
[4] "GtkButton"           "GtkBin"              "GtkContainer"
[7] "GtkWidget"           "GtkObject"           "GInitiallyUnowned"
[10] "GObject"             "RGtkObject"
```

```
$greater
<pointer: 0x2cea2b0>
attr(,"interfaces")
[1] "GtkBuildable"          "AtkImplementorIface"
attr(,"class")
[1] "GtkRadioButton"      "GtkCheckButton"      "GtkToggleButton"
[4] "GtkButton"           "GtkBin"              "GtkContainer"
[7] "GtkWidget"           "GtkObject"           "GInitiallyUnowned"
[10] "GObject"             "RGtkObject"
```

```
w <- gtkWindow();
w['title'] <- "Radio group example"
g <- gtkVBox(); w$add(g)
supply(rev(radiogp$getGroup()), gtkBoxPackStart, object = g)
```

```
[[1]]
NULL
```

```
[[2]]
NULL
```

```
[[3]]
NULL
```

Combo boxes

The combo box is a more space efficient alternative to radio buttons and is better suited for when there are a large number of options. A basic, text-only `GtkComboBox` is constructed by `gtkComboBoxNewText`. Later we will discuss more complicated combo boxes, where an underlying data model is manipulated.

For the basic combo box, items may be added in a few different ways. The methods `appendText` and `prependText` add a text item to the end or

beginning, respectively. A text item is inserted at an arbitrary position in the list with the `insertText`.

The currently selected value is specified by index with the method `setActive` and returned by `getActive`. The index, as usual, is 0-based, and in this case, a value of `-1` indicates that no value is selected. The `getActiveText` method can be used to retrieve the text shown by the basic combo box.

Although combo boxes are much more space efficient than radio buttons, it can be difficult to use a combo box when there are a large number of selections. The `setWidth` method specifies the preferred number of columns for displaying the items.

The main signal to connect to is `changed` which is emitted when the active item is changed either by the user or the programmer through the `getActive` method.

Example 7.11: Combo box

A simple combo box may be produced as follows:

```
vals <- c("two.sided", "less", "greater")
cb <- gtkComboBoxNewText()
sapply(vals, gtkComboBoxAppendText, object = cb)
```

```
$two.sided
NULL
```

```
$less
NULL
```

```
$greater
NULL
```

```
cb$setActive(0) # first one
gSignalConnect(cb, "changed",
  f = function(w, ...) {
    i <- w$getActive() + 1 # shift index
    if(i == 0)
      cat("No value selected\n")
    else
      cat("Value is", w$getActiveText(), "\n")
  })
```

```
changed
166
attr(,"class")
[1] "CallbackID"
```

```
w <- gtkWindow(show=FALSE)
w['title'] <- "Combobox example"
```

```
w$add(cb)
w$show()
```

Sliders

The slider widget and spin button widget allow selection from a regularly spaced, semi-continuous list of values.

The slider widget is called `GtkScale` and may be oriented either horizontally or vertically. This depends on the constructor: `gtkHScale` or `gtkVScale`. The user must specify the minimum, maximum and step values for the scale. This set of values is formally represented by the `GtkAdjustment` structure. Ordinarily, it is not necessary to construct a `GtkAdjustment` explicitly. Instead, the constructors accept the numeric arguments `min`, `max`, and `step`.

The underlying `GtkAdjustment` serves as the data model for the slider. Multiple sliders can be synchronized by attaching to the same adjustment object.

The methods `getValue` and `setValue` can be used to get and set the value of the widget. Values are clamped to the bounds defined by the adjustment.

A few properties define the appearance of the slider widget. The `digits` property controls the number of digits after the decimal point. The property `draw-value` toggles the drawing of the selected value near the slider. Finally, `value-pos` specifies where this value will be drawn using values from `GtkPositionType`. The default is `top`.

Callbacks can be assigned to the `value-changed` signal, which is emitted when the slider is moved.

Example 7.12: A slider controlling histogram bin selection

A simple mechanism to make a graph interactive is to redraw the graph whenever a slider, controlling a plot parameter, is changed. The following shows how this can be achieved.

```
library(lattice)
w <- gtkWindow(); w$setTitle("Histogram bin selection")
slider <- gtkHScale(min = 1, max = 100, step = 1)
slider$setValue(10) # initial val.
slider['value-pos'] <- "bottom"
w$add(slider)
drawHistogram <- function(val) print(histogram(x, nint = val))
gSignalConnect(slider, "value-changed",
               f = function(w, ...) {
                 val <- w$getValue()
                 drawHistogram(val)
               })
```

```
value-changed
```



```

174
attr(,"class")
[1] "CallbackID"

```

```

x <- rnorm(100) # the data
drawHistogram(slider$getValue()) # initial graphic

```

Spin buttons

The spin button widget is very similar to the slider widget, conceptually and in terms of the GTK+ API. Spin buttons are constructed with `gtkSpinButton`. As with sliders, this constructor requires specifying adjustment values, either as a `GtkAdjustment` or individually.

As with sliders, the methods `getValue` and `setValue` get and set the widget's value. The property `snap-to-ticks` can be set to `TRUE` to force the new value to belong to the sequence of values in the adjustment. The `wrap` property indicates if the sequence will “wrap” around at the bounds.

The value-changed signal is emitted when the spin button is changed, as with sliders.

Example 7.13: A range widget

This example shows how to make a range widget that combines both the slider and spinbutton to choose a single number. Such a widget is popular, as the slider is better at large changes and the spin button better at finer changes. In GTK+ we use the same `GtkAdjustment` model, so changes to one widget propagate without effort to the other.

Were this written as a function, an R user might expect the arguments to match those of `seq`:

```

from <- 0; to <- 100; by <- 1

```

The slider is drawn without a value, as the value is already displayed by the spin button. The call to `gtkHScale` implicitly creates an adjustment for the slider. The spin button is created with the same adjustment.

```

slider <- gtkHScale(min=from, max=to, step=by)
slider['draw-value'] <- FALSE
adjustment <- slider$getAdjustment()
spinbutton <- gtkSpinButton(adjustment = adjustment)

```

Our layout places the two widgets in a horizontal box container with the slider set to expand into the available space, but not the spinbutton.

```

g <- gtkHBox()
g$packStart(slider, expand=TRUE, fill=TRUE, padding=5)
g$packStart(spinbutton, expand=FALSE, padding=5)

```

7.6 Containers

In Section ??, we presented `GtkBox` and `GtkAlignment`, the two most useful layout containers in GTK+. This section introduces some other important containers. These include the merely decorative `GtkFrame`; the interactive `GtkExpander`, `GtkPaned` and `GtkNotebook`; and the grid-style layout container `GtkTable`. All of these widgets are derived from `GtkContainer`, and so share methods like `add`, which adds a child.

Framed containers

The `gtkFrame` function constructs a container that draws a decorative, labeled frame around its single child. This is useful for visually segregating a set of conceptually related widgets from the rest of the GUI. The optional `label` argument specifies the label text, stored in the `label` property. The `setLabelAlign` aligns the label relative to the frame. Frames have a decorative shadow whose type, a value of `GtkShadowType`, is stored in the `shadow-type` property.

Expandable containers

The `GtkExpander` widget provides a button that hides and shows a single child upon demand. This is often an effective mechanism for managing screen space. Expandable containers are constructed by `gtkExpander`. Use `gtkExpanderNewWithMnemonic` if a mnemonic is desired. The label text can be passed to the constructor or set later with the `setLabel` method. The `expanded` property, which can be accessed with `getExpanded` and `setExpanded`, represents the visible state of the widget. When the `expanded` property changes, the `activate` signal is emitted.

Notebooks

The `gtkNotebook` constructor creates a notebook container, a widget that displays an array of buttons resembling notebook tabs. Each tab corresponds to a widget, and when a tab is selected, its widget is made visible, while the others are hidden. If `GtkExpander` is like a check button, `GtkNotebook` is like a radio button group.

The current page number is stored in the `page` property. The total number of pages is returned by `getNPages`. The default position of the notebook tabs is on the top, ordered from left to right. The property `tab-pos` represents the tab position with a value from `GtkPositionType`: "left", "right", "top", or "bottom".

Adding pages to a notebook New pages can be added to the notebook with the `InsertPage` method, which takes the widget associated with the page, the 0-based insertion position (defaults to last), as well as a widget, such as a `GtkLabel` instance, not a string, to label the tab. This allows for more complicated tabs, such as a box container with a label and close icon. The `setTabLabelText` method is a convenience for setting a label as text. To use this method, the child widget is needed, which can be retrieved with the `[[` method or the `getNthPage` method. Both are a shortcut around retrieving all of the children as a list through `getChildren`.

Manipulating pages Methods that manipulate pages operate on the page number. To map from the child widget to the page number, use the method `pageNum`. A given page can be raised with the `setCurrentPage` method. Incremental movements are possible through the methods `nextPage` and `prevPage`.

Pages can be reordered using the `reorderChild`, although it is usually desirable to allow the user to reorder pages. The `setTabReorderable` enables drag and drop reordering for a specific tab. It is also possible for the user to drag and drop pages between notebooks. Pages can be deleted using the method `removePage`.

Managing Many Pages By default, a notebook will request enough space to display all of its tabs. If there are many tabs, space may be wasted. `GtkNotebook` solves this with the scrolling idiom. If the property `scrollable` is set to `TRUE`, arrows will be added to allow the user to scroll through the tabs. In this case, the tabs may become difficult to navigate. Setting the `enable-popup` property to `TRUE` enables a right-click popup menu listing all of the tabs for direct navigation.

Signals The notebook widget emits signals when pages are toggled, added, removed, and reordered. The most useful is likely to be `switch-page`, which is emitted when the current page is changed.

Example 7.14: Adding a page with a close button

A familiar element of notebooks in many web browsers is a tab close button. The following defines a new method `insertPageWithCloseButton` that will use the themeable stock close icon. The callback passes both the notebook and the page through the data argument, so that the proper page can be deleted.

```
gtkNotebookInsertPageWithCloseButton <-
  function(object, child, label.text="", position=-1) {
    label <- gtkHBox()
    label$packStart(gtkLabel(label.text))
```

```
icon <- gtkImage(pixbuf = object$renderIcon("gtk-close", "button"))
closeButton <- gtkButton()
closeButton$setImage(icon)
label$packEnd(closeButton)
ID <- gSignalConnect(b,"clicked",
                    function(userData, b, ...) {
                        nb <- userData$nb
                        page <- userData$page
                        nb$removePage(nb$pageNum(page))
                    },
                    data = list(nb=object, page=child),
                    user.data.first=TRUE)
object$insertPage(child, label, position)
}
```

We now show a simple usage of a notebook.

```
w <- gtkWindow()
nb <- gtkNotebook(); w$add(nb)
nb$setScrollable(TRUE)
nb$insertPageWithCloseButton(gtkButton("hello"),
                             label.text="page 1")
```

```
[1] 0
```

```
nb$insertPageWithCloseButton(gtkButton("world"),
                             label.text="page 2")
```

```
[1] 1
```

Scrollable windows

The `GtkExpander` and `GtkNotebook` widgets support efficient use of screen real estate. However, when a widget is always too large to fit in a GUI, partial display is necessary. A `GtkScrolledWindow` supports this by providing scrollbars for the user to adjust the visible region of a single child. The range, step and position of `GtkScrollbar` are controlled by an instance of `GtkAdjustment`, just as with the slider and spin button.

The constructor `gtkScrolledWindow` creates a `GtkScrolledWindow` instance. By default, the horizontal and vertical adjustments are automatically determined, although they may be overridden by the programmer.

The widget in a scrolled window must know how to display only a part of itself, i.e., it must be scrollable. Some widgets, including `GtkTreeView` and `GtkTextView`, have native scrolling support. Other widgets must be embedded within the proxy `GtkViewport`. The `GtkScrolledWindow` convenience method `addWithViewport` allows the programmer to skip the `GtkViewport` step.

The properties `hscrollbar-policy` and `vscrollbar-policy` determine when the scrollbars are drawn. By default, they are always drawn. The "automatic" value from the `GtkPolicyType` enumeration draws the scrollbars only if needed, i.e, the child widget requests more space than can be allocated. The `setPolicy` method allows both to be set at once, as in the following example.

Example 7.15: Scrolled window example

This example shows how to display a long list of values with scrolled windows. The tree view widget can also do this, but here we can very easily customize the display of each value. In the example, we simply locate where a label is placed.

```
g <- gtkVBox(spacing=0)
sapply(state.name, function(i) {
  l <- gtkLabel(i)
  l['xalign'] <- 0; l['xpad'] <- 10
  g$packStart(l, expand=TRUE, fill=TRUE)
})
```

```
$Alabama
NULL

$Alaska
NULL

$Arizona
NULL

$Arkansas
NULL

$California
NULL

$Colorado
NULL

$Connecticut
NULL

$Delaware
NULL

$Florida
NULL
```

7. RGtk2: BASIC COMPONENTS

\$Georgia
NULL

\$Hawaii
NULL

\$Idaho
NULL

\$Illinois
NULL

\$Indiana
NULL

\$Iowa
NULL

\$Kansas
NULL

\$Kentucky
NULL

\$Louisiana
NULL

\$Maine
NULL

\$Maryland
NULL

\$Massachusetts
NULL

\$Michigan
NULL

\$Minnesota
NULL

\$Mississippi
NULL

\$Missouri
NULL

\$Montana
NULL

\$Nebraska
NULL

\$Nevada
NULL

\$ 'New Hampshire '
NULL

\$ 'New Jersey '
NULL

\$ 'New Mexico '
NULL

\$ 'New York '
NULL

\$ 'North Carolina '
NULL

\$ 'North Dakota '
NULL

\$Ohio
NULL

\$Oklahoma
NULL

\$Oregon
NULL

\$Pennsylvania
NULL

\$ 'Rhode Island '
NULL

\$ 'South Carolina '
NULL

\$ 'South Dakota '

```
NULL

$Tennessee
NULL

$Texas
NULL

$Utah
NULL

$Vermont
NULL

$Virginia
NULL

$Washington
NULL

$`West Virginia`
NULL

$Wisconsin
NULL

$Wyoming
NULL
```

The scrolled window has just two basic steps in its construction. Here we specify never using a scrolled window for the vertical display.

```
sw <- gtkScrolledWindow()
sw$setPolicy("never","automatic")
sw$addWithViewport(g)                                # just "Add" for text, tree, ...

w <- gtkWindow(show=FALSE)
w$setTitle("Scrolled window example")
w$setSizeRequest(-1, 300)
w$add(sw)
w$show()
```

Divided containers

The `gtkHPaned` and `gtkVPaned` create containers that contain two widgets, arranged horizontally or vertically and separated by a handle. The user may adjust the position of the handle to apportion the allocation between the widgets.

The two children may be added two different ways. The methods `pack1` and `pack2` have arguments `resize`, whether the child expands with the parent, and `shrink`, whether the widget is allowed to shrink. The methods `add1` and `add2` add children such that both are allowed to shrink and only the second widget expands. After children are added, they can be referenced from the container through the `getChild1` and `getChild2` methods.

The screen position of the handle can be set with the `setPosition` method. The properties `min-position` and `max-position` can be used to convert a percentage into a screen position. The `move-handle` signal is emitted when the gutter position is changed.

Tabular layout

The `gtkTable` constructor produces a container for laying out objects in a tabular format. The container sets aside cells in a grid, and a child widget may occupy one or more cells. The `homogeneous` argument can be used to make all cells homogeneous in size. Otherwise, each column and row can have a different size. At the time of construction, the number rows and columns for the table may be specified with the `rows` and `columns` arguments. After construction, the `Resize` method can be used to resize these values.

Child widgets are added to this container through the `attach` method. Its first argument, `child`, is the child widget. This widget can span more than one cell. The arguments `left.attach` and `right.attach` specify the horizontal bounds of the child in terms of its left column and right column, respectively. Analogously, `top.attach` and `bottom.attach` define the vertical bounds. By default, the widgets will expand into and fill the available space, much as if `expand` and `fill` were passed as `TRUE` to `packStart` (see Section 7.2). There is no padding between children by default. Both the resizing behavior and padding may be overridden by specifying additional arguments to `attach`.

The child properties `xalign` and `yalign` specify the alignment of child widgets within their allocated space. These behave as with `GtkAlignment`.

Example 7.16: Dialog layout

This example shows how to layout some controls for a dialog with some attention paid to how the widgets are aligned and how they respond to resizing of the window.

Our basic GUI is a table with 4 rows and 2 columns.

```
w <- gtkWindow(show=FALSE)
w$setTitle("example of gtkTable and attaching")
tbl <- gtkTable(rows=4, columns=2, homogeneous=FALSE)
w$add(tbl)
```

We define our widgets first then deal with their layout.

7. RGtk2: BASIC COMPONENTS

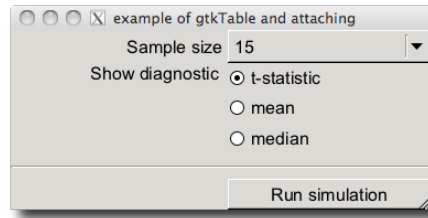


Figure 7.7: A basic dialog using a `gtkTable` container for layout.

```
l1 <- gtkLabel("Sample size")
w1 <- gtkComboBoxNewText()
QT <- sapply(c(5, 10, 15, 30), function(i) w1$appendText(i))
l2 <- gtkLabel("Show diagnostic ")
w2 <- gtkVBox()
rb <- list()
rb[["t"]] <- gtkRadioButton(label="t-statistic")
for(i in c("mean","median")) rb[[i]] <- gtkRadioButton(rb, label=i)
QT <- sapply(rb, function(i) w2$packStart(i))
w3 <- gtkButton("Run simulation")
```

The basic `AttachDeafults` method will cause the widgets to expand when resized, which we want to control here. As such we use `Attach`. To get the control's label to center align yet still have some breathing room we set its `xalign` and `xpad` properties. For the combobox we avoid using "expand" as otherwise it resizes to fill the space allocated to the cell in the y direction.

```
tbl$attach(l1, left.attach=0,1, top.attach=0,1, yoptions="fill")
l1["xalign"] <- 1; l1["xpad"] <- 5
tbl$attach(w1, left.attach=1,2, top.attach=0,1, xoptions="fill", yoptions="fill")
```

We use "expand" here to attach the radio group, so that it expands to fill the space. The label has its `yalign` property set, so that it stays at the top of the cell, not the middle.

```
tbl$attach(l2, left.attach=0,1, top.attach=1,2, yoptions="fill")
l2["xalign"] <- 1; l2["yalign"] <- 0; l2["xpad"] <- 4
tbl$attach(w2, left.attach=1,2, top.attach=1,2, xoptions=c("expand", "fill"))
```

A separator with a bit of padding provides a visual distinction between the controls and the button to initiate an action.

```
tbl$attach(gtkHSeparator(), left.attach=0,2, top.attach=2,3, ypadding=10, yoptions="fill")
tbl$attach(w3, left.attach=1,2, top.attach=3,4, xoptions="fill", yoptions="fill")
```

Finally, we use the `ShowAll` method so that it propagates to the combobox.

```
w$showAll() # propagate to combo
```

7.7 Drag and drop

GTK+ has mechanisms to provide drag and drop facilities for widgets. To setup drag and drop actions requires setting a widget to be a source for a drag request, and setting a widget to be a target for a drop action, and assigning callbacks to respond to certain signals. Only widgets which can receive signals will work for drag and drop, so to drag or drop on a label, say, an event box must be used.

We illustrate how to set up the dragging of a text value from one widget to another. Much more complicated examples are possible, but we do not pursue it here.

When a drag and drop is initiated, different types of data may be transferred. GTK+ allows the user to specify a target type. Below, we define target types for text and pixmap objects. These give numeric IDs for lookup purposes.

```
TARGET.TYPE.TEXT    <- 80
TARGET.TYPE.PIXMAP  <- 81
```

We use of these to make different types of objects that can be dragged.

```
widgetTargetTypes <- list(
  ## target — string representing the drag type. MIME type used.
  ## flag delimiting drag scope. 0 — no limit
  ## info — application assigned value to identify
  text = gtkTargetEntry("text/plain", 0, TARGET.TYPE.TEXT),
  pixmap = gtkTargetEntry("image/x-pixmap", 0, TARGET.TYPE.PIXMAP)
)
```

A drag source A widget that can have a value dragged from it is a drag source. It is specified by calling `gtkDragSourceSet`. This function has arguments object for the widget we are making a source, `start.button.mask` to specify which mouse buttons can initiate the drag, targets to specify the target type, and actions to indicate which of the `GdkDragAction` types is in effect, for instance copy or move.

When a widget is a drag source, it sends the data being dragged in response to the `drag-data-get` signal using a callback. The signature of this callback is important, although we only use the `selection` argument, as this is assigned the text that will be the data passed to the target widget. (Text, as we are passing text information.)

```
w <- gtkWindow(); w['title'] <- "Drag Source"
dragSourceWidget <- gtkButton("Drag me")
w$add(dragSourceWidget)
gtkDragSourceSet(dragSourceWidget,
  start.button.mask=c("button1-mask", "button3-mask"),
  targets=widgetTargetTypes[["text"]],
```

```
actions="copy") ## can also be any of GdkDragAction
ID <-
  gSignalConnect(dragSourceWidget, "drag-data-get",
    f=function(widget, context,
      selection, targetType, eventTime) {
      ## customize this to set the text
      selection$setText(str="some value")
    })
```

Drop target To make a widget a drop target, we call `gtkDragDestSet` on the object with the argument flags for specifying the actions GTK+ will perform when the widget is dropped on. We use the value "all" for "motion", "highlight", and "drop". The targets argument matches the type of data being allowed, in this case text. Finally, the value of action specifies what `GdkDragAction` should be sent back to the drop source widget. If the action was "move" then the source widget emits the `drag-data-delete` signal, so that a callback can be defined to handle the deletion of the data.

```
w <- gtkWindow(); w['title'] <- "Drop Target"
dropTargetWidget <- gtkButton("Drop here")
w$add(dropTargetWidget)
gtkDragDestSet(dropTargetWidget,
  flags="all",
  targets=widgetTargetTypes[["text"]],
  actions="copy"
)
```

When data is dropped, the widget emits the `drag-data-received`. The data is passed through the `selection` argument. The `context` argument is a `gdkDragContext`, containing information about the drag event. The `x` and `y` arguments are integer valued and pass in the position in the widget where the drop occurred. In the example below, we see that text data is passed to this function in raw format, so it is converted with `rawToChar`.

```
ID <-
  gSignalConnect(dropTargetWidget, "drag-data-received",
    f=function(dropTargetWidget,
      context, x, y,
      selection, targetType, eventTime) {
      dropdata <- selection$getText()
      if(class(dropdata)[1] == "raw")
        val <- paste(rawToChar(dropdata), sep="")
      else
        val <- paste(dropdata, sep="")
      print(val) ## some action
    })
```

7.8 Graphics

The cairoDevice package

The package `cairoDevice` is an R graphics device based on the Cairo graphics library. It is cross-platform and supports alpha-blending and antialiasing. Through its support for the `getGraphicsEvent` function, it is currently the most interactive cross-platform graphics device.

`RGtk2` and `cairoDevice` are integrated through the `asCairoDevice` function. If a `GtkDrawingArea`, `GdkDrawable`, Cairo context, or `GtkPrintContext` is passed to `asCairoDevice`, an R graphics device will be initialized that targets its drawing to the object. For simply displaying graphics in a GUI, the `GtkDrawingArea` is the best choice. For more complex use cases, such as compositing a layer above or below the R graphic, one should pass an off-screen `GdkDrawable`, like a `GdkPixmap`, or a Cairo context. The off-screen drawing can then be composited with other images when displayed. Finally, passing a `GtkPrintContext` to `asCairoDevice` allows printing R graphics through the GTK+ printing dialogs.

RGtk2: Widgets Using Models

Many widgets in GTK+ use the model, view, controller (MVC) paradigm. For most, like the button, the MVC pattern is implicit; however, widgets that primarily display data explicitly incorporate the MVC pattern into their design. The data model is factored out as a separate object, while the widget plays the role of the view and controller. The MVC approach adds a layer of complexity but facilitates the display of the dynamic data in multiple, coordinated views.

8.1 Display of tabular data

Widgets that display lists, tables and trees are all based on the same basic data model, `GtkTreeModel`. Although its name suggests a hierarchical structure, `GtkTreeModel` is also tabular. We first describe the display of an R data frame in a list or table view. The display of hierarchical data, as well as further details of the `GtkTreeModel` framework, are treated subsequently.

Loading a data frame

As an interface, `GtkTreeModel` may be implemented in any number of ways. GTK+ provides simple in-memory implementations for hierarchical and non-hierarchical data. For improved speed, convenience and familiarity, RGtk2 includes a custom `GtkTreeModel` implementation called `classRGtkDataFrame`, which is based on an R data frame. For non-hierarchical data, this is usually the model of choice, so we discuss it first.

R uses data frames to hold tabular data, where each column is of a certain class, and each row is related to some observational unit. This fits the structure of `GtkTreeModel` when there is no hierarchy. As such it is natural to have a means to map a data frame into a store for a tree view. `RGtkDataFrame` implements `GtkTreeModel` to perform this role and is constructed with the `rGtkDataFrame` function. Populating a `RGtkDataFrame` is much faster than a GTK+ model, because the data is taken directly from the data frame, without any copying or need to add each row individually in a slow R loop.

The constructor takes a data frame as an argument. The column classes are important, so even if this data frame is empty, the user should specify the desired column classes upon construction.

An object of class `RGtkDataFrame` supports the familiar S3 methods `[], [<-, dim,` and `as.data.frame`. The `[<-` method does not have quite the same functionality as it does for a data frame. Columns can not be removed by assigning values to `NULL`, and column types should not be changed. These limitations are inherent in the design of GTK+: columns may not be removed from `GtkTreeModel`, and views expect the data type to remain the same.

Example 8.1: Defining and manipulating a `RGtkDataFrame`

The basic data frame methods are similar.

```
data(Cars93, package="MASS")           # mix of classes
model <- rGtkDataFrame(Cars93)
model[1, 4] <- 12
model[1, 4]                             # get value
```

```
[1] 12
```

Factors are treated differently from character values, as is done with data frames, so assignment to a factor must be from one of the possible levels.

The data frame combination functions `rbind` and `cbind` are unsupported, as they would create a new data model, rather than modify the model in place. Thus, one should add rows with `appendRows` and add columns with `appendColumns` (or sub-assignment, `[<-`).

The `setFrame` method replaces the underlying data frame.

```
model$setFrame(Cars93[1:5, 1:5])
```

```
NULL
```

Replacing the data frame is the only way to remove rows, as this is not possible with conventional data frame sub-assignment interface. Removing columns or changing their types remains impossible. The new data frame cannot contain more columns and rows than the current one. If the new data frame has more rows or columns, then the appropriate append method should be used first.

Displaying data as a list or table

`GtkTreeView` is the primary view of `GtkTreeModel`. It serves as the list, table and tree widget in GTK+. A tree view is essentially a container of columns, where every column has the same number of rows. If the view has a single column, it is essentially a list. If there are multiple columns, it is a table. If the rows are nested, it is a tree table, where every node has values on the same columns.

A tree view is constructed by `gtkTreeView`.

```
view <- gtkTreeView(model)
```

Usually, as in the above, the model is passed to the constructor. Otherwise, the model may be accessed with `setModel` and `getModel`.

A newly created tree view displays zero columns, regardless of the number of columns in the model. Each column, an instance of `GtkTreeViewColumn`, must be constructed, inserted into the view and instructed to render content based on one or more columns in the data model:

```
vc <- gtkTreeViewColumn()
vc$setTitle("Manufacturer")
cr <- gtkCellRendererText()
vc$packStart(cr)
vc$addAttribute(cr, "text", 0)
view$insertColumn(vc, 0)
```

```
[1] 1
```

A column with the title “Manufacturer” is inserted at the first, 0-based, position. For displaying a simple data frame, we only need to render text. Each row in a column consists of one or more cells, managed in a layout. The number of cells and how each cell is rendered is uniform down a column. As an implementation of `GtkCellLayout`, `GtkTreeViewColumn` delegates the responsibility of rendering to one or more `GtkCellRenderer` objects. The cell renderers are packed into the column, which behaves much like a box container. Rendering of text cells is the role of `GtkCellRendererText`; we create an instance with `gtkCellRendererText`. There are several properties that control how the text is rendered. A so-called *attribute* links a model column to a renderer property. The most important property is `text`, the text itself. In the example, we bind the `text` property to the first (0-indexed) column in the model.

`GtkTreeView` provides the `insertColumnWithAttributes` convenience method to perform all of these steps with a single call. We invoke it to add a second column in our view:

```
view$insertColumnWithAttributes(position = -1, title = "Model",
                                cell = gtkCellRendererText(), text = 1)
```

```
[1] 2
```

The `-1` passed as the first argument indicates that the column should be appended. Next, we specify the column title, a cell renderer, and an attribute that links the `text` renderer property to the second column in the model. In general, any number of attributes may be defined after the third argument. We will use the above idiom in all of the following examples, as it is much more concise than performing each step separately.

To display the entire Cars93 data frame, we insert a view column for every column in the data frame. Here, we reconstruct the view, inserting a view column for every column in the data frame, i.e., the model.

```
view <- gtkTreeView(model)
mapply(view$insertColumnWithAttributes, -1, colnames(model),
       list(gtkCellRendererText()), text = seq_len(ncol(model)) - 1)
```

```
[1] 1 2 3 4 5
```

Although it was relatively easy to create a `GtkTreeModel` for the data frame using `RGtkDataFrame`, the complexity of `GtkTreeView` complicates the task of displaying the data frame in a simple, textual table. When this is all that is necessary, one might consider `gdataframe` from `gWidgets`. For those who wish to render text in each row differently (e.g., in a different color) or fill cells with images, check boxes, progress bars and the like, direct use of the `GtkTreeView` API is required.

Manipulating view columns The `GtkTreeView` widget is essentially a collection of columns. Columns are added to the tree view with the methods `insertColumn` or, as shown above, `insertColumnWithAttributes`. A column can be moved with the `moveColumnAfter` method, and removed with the `removeColumn` method. The `getColumns` method returns a list containing all of the tree view columns.

There are several properties for controlling the behavior and dimensions of a `GtkTreeViewColumn` instance. The property `"resizable"` determines whether the user can resize a column, by dragging with the mouse. The size properties `"width"`, `"min-width"`, and `"fixed-width"` control the size. The visibility of the column can be adjusted through the `setVisible` method.

Additional Features Tree views have several special features, including sorting, incremental search and drag-n-drop reordering. Sorting is discussed in Section ???. To turn on searching, `enable-search` should be `TRUE` (the default) and the `search-column` property should be set to the column to be searched. The tree view will popup a search box when the user types control-f. To designate an arbitrary text entry widget as the search box, call `setSearchEntry`. The entry can be placed anywhere in the GUI. Columns are always reorderable by drag and drop. Reordering rows through drag-and-drop is enabled by the `reorderable` property.

Aesthetic properties `GtkTreeView` is capable of rendering some visual guides. The `rules-hint`, if `TRUE`, will instruct the theme to draw rows in alternating colors. To show grid lines, set `enable-grid-lines` to `TRUE`.

Accessing GtkTreeModel

Although `RGtkDataFrame` provides a familiar interface for manipulating the data in a `GtkTreeModel`, it is often necessary to directly interact with the GTK+ API, such as when using another type of data model or interpreting user selections. There are two primary ways to index into the rows of a tree model: paths and iterators.

To index directly into an arbitrary row, a `GtkTreePath` is appropriate. For a list, a tree path is essentially the row number, 0-based; for a tree it is a sequence of integers referring to the offspring index at each level. The sequence of integers may be expressed as either a numeric/integer vector or a string, using `gtkTreePathNewFromIndices` or `gtkTreePathNewFromString`, respectively. For a flat list model, there is only one integer in the sequence:

```
secondRow <- gtkTreePathNewFromIndices(2)
```

Referring to a row in a hierarchy is slightly more complex:

```
abcPath <- gtkTreePathNewFromIndices(c(1, 3, 2))
abcPath <- gtkTreePathNewFromString("1:3:2")
```

In the above, both paths refer to the second child of the third child of the first top-level node. To recover the integer or string representation of the path, use `getIndices` or `toString`, respectively.

The second means of row indexing is through an iterator, `GtkTreeIter`, which is better suited for traversing a model. While a tree path is an intuitive, transparent row index, an iterator, by contrast, is an opaque index that is efficiently incremented. It is probably most common for a model to be accessed in an iterative manner, so all of the data accessor methods for `GtkTreeModel` expect `GtkTreeIter`, not `GtkTreePath`. The GTK+ designers imagined that the typical user would obtain an iterator for the first row and to visit each row in sequence:

```
### JV HAD issues with thi
iter <- model$getIterFirst()
manufacturer <- character()
while(iter$retval) {
  manufacturer <- c(manufacturer, model$get(iter$iter, 0)[[1]])
  model$iterNext(iter$iter)
}
```

In the above, we recover the manufacturer column from the `Cars93` data frame. Whenever a `GtkTreeIter` is returned by a `GtkTreeModel`, the return value in R is a list of two components: `retval`, a logical indicating whether the iterator is valid, and `iter`, the pointer to the underlying C data structure. The call to `get` also returns a list, with an element for each column index passed as an argument. The method `iterNext` updates the passed iterator in place, i.e., by reference, to point to the next row. Thus, no new iterator is

returned. This is unfamiliar behavior in R. Instead, the method returns a logical value indicating whether the iterator is still valid, i.e. `FALSE` is returned if no next row exists.

It is clear that the above usage is designed for languages like C, where multiple return values are conveniently passed by reference parameters. The iterator design also prevents the use of the `apply` functions, which are generally preferred over the `while` loop for reasons of performance and clarity. An improvement would be to obtain the number of children, generate the sequence of row indices and access the row for each index:

```
nrows <- model$iterNChildren(NULL)
manufacturer <- sapply(seq(nrows), function(i) {
  iter <- model$iterNthChild(NULL, i)
  model$get(iter$iter, 0)[[1]]
})
```

Here we use `NULL` to refer to the virtual root node that sits above the rows in our table. Unfortunately, this usage too is unintuitive and slow, so the benefits of `RGtkDataFrame` should be obvious.

One can convert between the two representations. The method `getIter` on `GtkTreeModel` returns an iterator for a path. A shortcut from the string representation of the path to an iterator is `getIterFromString`. The path pointed to by an iterator is returned by `getPath`.

One might note that `GtkTreeIter` is created and managed by the model, while `GtkTreePath` is model independent. It is not possible to use iterators across models or even across modifications to a model. After a model changes, an iterator is invalid. A tree path may still point to a valid row, though it will not in general be the same row from before the change. To refer to the same row across tree model changes, use a `GtkTreeRowReference`, not discussed further.

Selection

There are multiple modes of user interaction with a tree view: if the cells are not editable, then selection is the primary mode. A single click selects the value, and a double click is often used to initiate an action. If the cells are editable, then a double click or a click on an already selected row will initiate editing of the content. Editing of cell values is a complex topic and is handled by derivatives of `GtkCellRenderer`, see Section 8.1. Here, we limit our discussion to selection of rows.

GTK+ provides the class `GtkTreeSelection` to manage row selection. Every tree view has a single instance of `GtkTreeSelection`, returned by the `getSelection` method.

The usage of the selection object depends on the selection mode, i.e., whether multiple rows may be selected. The mode is configured with the

`setMode` method, with values from `GtkSelectionMode`, including "multiple" for allowing more than one row to be selected and "single" for limiting selections to a single row, or none.

When only a single selection is possible, the method `getSelected` returns the selected row as list, with components `retval` to indicate success, `model` pointing to the tree model and `iter` representing an iterator to the selected row in the model.

```
model <- rGtkDataFrame(mtcars)
view <- gtkTreeView(model)
selection <- view$getSelection()
selection$setMode("single")
```

```
[1] 1
```

If this tree view is shown and a selection made, this code will return the value in the first column:

```
selection$selectPath(gtkTreePathNewFromIndices(3)) # set
#
curSel <- selection$getSelected() # retrieve selection
with(curSel, model$getValue(iter, 0)$value) # model, iter
```

```
[1] 21.4
```

When multiple selection is permitted, then the method `getSelectedRows` returns a list with components `model` pointing to the model, and `retval`, a list of tree paths.

For example, we can change the selection mode as follows.

```
selection$setMode("multiple")
```

This code will print the selected values in the first column (we have selected the first three rows):

```
curSel <- selection$getSelectedRows()
if(length(curSel$retval)) {
  rows <- sapply(curSel$retval, gtkTreePathGetIndices) + 1L
  curSel$model[rows, 1]
}
```

```
[1] 21.0 22.8 21.4
```

To respond to a selection, connect to the changed signal on `GtkTreeSelection`. The signal itself does not contain any selection information; the selection object should be queried instead.

When a row is not editable, then the double-click event or a keyboard command triggers the row-activated signal for the tree view. The callback has arguments `tree.view` pointing to the widget that emits the signal, `path` storing a tree path of the selected row, and `column` containing the tree view

column. The column number is not returned. If that is of interest, it can be passed in via the user data argument, or matched against the children of the tree view through a command like

```
supply(view$getColumn(), function(i) i == column)
```

Sorting

A common GUI feature is sorting a table widget by column. By convention, the user clicks on the column header to toggle sorting. `GtkTreeView` supports this interaction, although the actual sorting occurs in the model. Any model that implements the `GtkTreeSortable` interface supports sorting. `RGtkDataFrame` falls into this category. When `GtkTreeView` is directly attached to a sortable model, it is only necessary to inform each view column of the model column to use for sorting when the header is clicked:

```
vc <- view$getColumn(0)
vc$setSortColumnId(0)
```

In the above, clicking on the header of the first view column, `vc`, will sort by the first model column. Behind the scenes, `GtkTreeViewColumn` will set its sort column as the sort column on the model, i.e.:

```
model$setSortColumnId(0, GtkSortType['ascending'])
```

Some models, however, do not implement `GtkTreeSortable`, such as `GtkTreeModelFilter`, introduced in the next section. Also, sorting a model permanently changes the order of its rows, which may be undesirable in some cases. The solution is to proxy the original model with a sortable model. The proxy obtains all of its data from the original model and reorders the rows according to the order of the sort column. GTK+ provides `GtkTreeModelSort` as a sorting proxy model.

```
store <- rGtkDataFrame(mtcars)
sorted <- gtkTreeModelSortNewWithModel(store)
view <- gtkTreeView(sorted)
view$insertColumnWithAttributes(0, "Click to sort", gtkCellRendererText(),
                               text = 0)
```

```
[1] 1
```

```
view$getColumn(0)$setSortColumnId(0)
```

When the user sorts the table, the underlying store will not be modified.

The default sorting function can be changed by calling the method `setSortFunc` on a sortable model. The following function shows how the default sorting might be implemented.

```
f <- function(model, iter1, iter2, user.data) {
  column <- user.data
  val1 <- model$getValue(iter1, column)$value
  val2 <- model$getValue(iter2, column)$value
  as.integer(val1 - val2)
}
sorted$setSortFunc(sort.column.id=0, sort.func=f, user.data=0)
# column
```

NULL

Filtering

The previous section introduced the concept of a proxy model in `GtkTreeModelSort`. Another common application of proxying is filtering. For filtering via a proxy model, GTK+ provides the `GtkTreeModelFilter` class. The basic idea is that an extra column in the base model stores logical values to indicate if a row should be visible. The index of that column is passed to the filter model, which provides only those rows where the filter column is `TRUE`.

```
df <- data.frame(col=letters[1:3], vis=c(TRUE, TRUE, FALSE))
store <- rGtkDataFrame(df)
filtered <- store$filter()
filtered$setVisibleColumn(1)          # 0-based
view <- gtkTreeView(filtered)
```

The constructor of the filter model is `gtkTreeModelFilter`, which, somewhat coincidentally, also works as a method on the base model, i.e., `model$filter()`. To retrieve the original model from the filter, call its `getModel` method. The method `setVisibleColumn` specifies which column in the model holds the logical values. The configured filter model may now be treated as any other tree model, including attachment to a `GtkTreeView`.

Example 8.2: Using filtering

This example shows how to use `GtkTreeModelFilter` to filter rows according to whether they match a value entered into a text entry box. The end result is similar to an entry widget with completion.

First, we create a data frame. The `VISIBLE` column will be added to the `rGtkDataFrame` instance to adjust the visible rows.

```
df <- data.frame(state.name)
df$VISIBLE <- rep(TRUE, nrow(df))
store <- rGtkDataFrame(df)
```

The filtered store needs to have the column specified that contains the logical values; in this example, it is the last column.

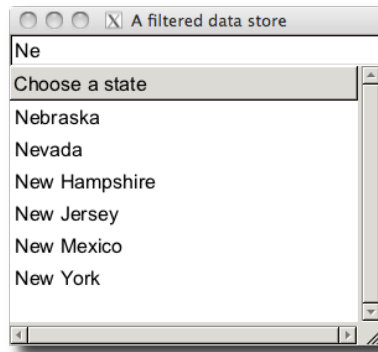


Figure 8.1: Example of a data store filtered by values typed into a text-entry widget.

```
filteredStore <- store$filter()
filteredStore$setVisibleColumn(ncol(df)-1)      # offset
view <- gtkTreeView(filteredStore)
```

Next, we create a basic view of a single column:

```
view$insertColumnWithAttributes(0, "Col", gtkCellRendererText(), text = 0)

[1] 1
```

An entry widget will be used to control the filtering. In the callback, we adjust the `VISIBLE` column of the `rgtkDataFrame` instance to reflect the rows to be shown. When `val` is an empty string, the result of `grepl` is `TRUE`, so all rows will be shown.

```
e <- gtkEntry()
gSignalConnect(e, "changed", function(w, data) {
  pattern <- w$getText()
  df <- data$getModel()
  values <- df[, "state.name"]
  df[, "VISIBLE"] <- grepl(pattern, values)
}, data=filteredStore)
```

```
changed
503
attr(,"class")
[1] "CallbackID"
```

Figure 8.1 shows the two widgets placed within a simple GUI.

Cell renderer details

The values in a tree model are rendered in a rectangular cell by the derivatives of `GtkCellRenderer`. Cell renderers are interactive, in that they also manage editing and activation of cells.

A cell renderer is independent of any data model. Its rendering role is limited to drawing into a specified rectangular region according to its current property values. An object that implements the `GtkCellLayout` interface, like `GtkTreeViewColumn` and `GtkComboBox` (see Section 8.3), associates a set of *attributes* with a cell renderer. An attribute is a link between an aesthetic property of a cell renderer and a column in the data model. When the `GtkCellLayout` object needs to render a particular cell, it configures the properties of the renderer with the values from the current model row, according to the attributes. Thus, the mapping from data to visualization depends on the class of the renderer instance, its explicit property settings, and the attributes associated with the renderer in the cell layout.

For example, to render text, a `GtkCellRendererText` is appropriate. The text property is usually linked via an attribute to a text column in the model, as the text would vary from row to row. However, the background color (the background property) might be common to all rows in the column and thus is set explicitly, without use of an attribute.

The base class `GtkCellRenderer` defines a number of properties that are common to all rendering tasks. The `xalign` and `yalign` properties specify the alignment, i.e., how to position the rendered region when it does not fill the entire cell. The `cell-background` property indicates the color for the entire cell background.

The rest of this section describes each type of cell renderer, as well as some advanced features.

Text cell renderers The `gtkCellRendererText` constructor is used to display text and numeric values. Numeric values in the model are shown as strings. The most important property is `text`, the actual text that is displayed. Other properties control the display of the text, such as the font family and size, the foreground and background colors, and whether to ellipsize or wrap the text if there is not enough space for display.

To display right-aligned text in a Helvetica font, the following could be used:

```
cr <- gtkCellRendererText()
cr['xalign'] <- 1                # default 0.5 = centered
cr['family'] <- "Helvetica"
```

The `wrap` attribute can be specified as `TRUE`, if the entries are expected to be long. There are several other attributes that can be changed.

When an attribute links the text property to a numeric column in the model, the property system automatically converts the number to its string representation. This occurs according to the same logic that R follows to print numeric values, so options like `scipen` and `digits` are considered. See the “Overriding attribute mappings” paragraph below for further customization.

Editable text renderers `GtkCellRendererCombo` and `GtkCellRendererSpin` allow editing a text cell with a combo box or spin button, respectively. Populating the combo box menu requires specifying two properties: `model` and `text-column`. The menu items are retrieved from the `GtkTreeModel` given by `model` at the column index given by `text-column`. If `has-entry` is `TRUE`, a combo box entry is displayed.

```
cr <- gtkCellRendererCombo()
store <- rGtkDataFrame(state.name)
cr['model'] <- store
cr['text-column'] <- 0
cr['editable'] <- TRUE # needed
```

The spin button editor is configured by passing a `GtkAdjustment` to the adjustment property.

Pixbuf cell renderers To display an image in a cell, `GtkCellRendererPixbuf` is appropriate. The image is specified through one of its properties: `stock-id`, a stock identifier; `icon-name`, the name of a themed icon; or `pixbuf`, an actual `GdkPixbuf` object. It is possible to store a `GdkPixbuf` in a `data.frame`, and thus an `RGtkDataFrame`, using a list:

```
library(RGtk2)
apple <- gdkPixbuf(filename = imagefile("apple-red.png"))[[1]]
floppy <- gdkPixbuf(filename = imagefile("floppybuddy.gif"))[[1]]
logo <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
rdf <- rGtkDataFrame(data.frame(image = I(list(apple, floppy, logo))))
view <- gtkTreeView(rdf)
view$insertColumnWithAttributes(0, "image", gtkCellRendererPixbuf(), pixbuf = 0)
```

```
[1] 1
```

```
win <- gtkWindow()
win$add(view)
```

Example 8.3: A widget for variable selection

This example shows a combination widget that is familiar from other statistics GUIs. It provides two tree views listing variable names and has arrows to move variable names from one side to the other. Often such widgets are used for specifying statistical models.

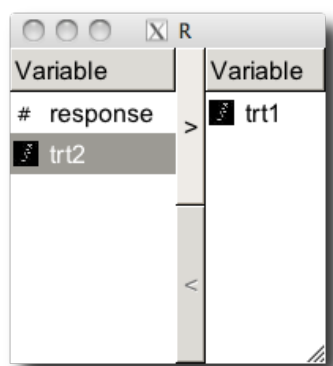


Figure 8.2: An example showing two tree views with buttons to move entries from one to the other. This is a common method for variable selection.

We will use Example 7.6, in particular its function `addToStockIcons`, to add some custom stock icons to identify the variable type.

```
fileNms <- c(factor = system.file("images", "factor.gif", package="gWidgets"),
             numeric = system.file("images", "numeric.gif", package="gWidgets"))
pixbufs <- lapply(fileNms, function(fn) gdkPixbuf(file = fn)[[1]])
addToStockIcons(pixbufs)
```

To keep track of the variables in the two tree views we use a single model. It has a column for all the variable names, a column for the icon, and two columns to keep track of which variable names are to be displayed in the respective tree views.

```
d <- data.frame(varNames=c("response", "trt1", "trt2"),
               stock.id=c("new-numeric", "new-factor", "new-factor"),
               leftView = rep(TRUE, 3),
               rightView = rep(FALSE, 3),
               stringsAsFactors=FALSE)
model <- rGtkDataFrame(d)
```

We will use a filtered data store to show each tree view. As the two tree views are identical, except for the rows that are displayed, we generate them with a function. The `vis.col` indicates which column in the `rGtkDataFrame` object contains the visibility information. Our tree view packs in both a `pixbuf` cell renderer and a text cell renderer.

```
makeView <- function(model, vis.col) {
  filteredModel <- model$filter()
  filteredModel$setVisibleColumn(vis.col - 1)
  tv <- gtkTreeView(filteredModel)
  tv$getSelection()$setMode("multiple")
}
```

```
##
cr <- gtkCellRendererPixbuf()
cr['xalign'] <- 1
tv$insertColumnWithAttributes(0, "Variable", cr, "stock-id" = 1)
vc <- tv$getColumn(0)
##
cr <- gtkCellRendererText()
vc$PackStart(cr, expand=TRUE)
cr['xalign'] <- 0
cr['xpad'] <- 5
vc$addAttribute(cr, "text", 0)

return(tv)
}
```

We now create the tree views:

```
views <- list()
views[["left"]] <- makeView(model,3)
views[["right"]] <- makeView(model,4)
selections <- lapply(views, gtkTreeViewGetSelection)
```

We need buttons to move the values left and right; these are stored in a list for convenience.

```
buttons <- list()
buttons[["fromLeft"]] <- gtkButton(">")
buttons[["fromRight"]] <- gtkButton("<")
```

Our basic GUI is shown in Figure 8.2 where the two tree views are placed side-by-side.

The key handler moves the selected value from one side to the other. A complication is that when the view is using filtering the selection returns values relative to the child model (the filtered one). In general, the methods `convertChildPathToPath` and `convertChildIterToIter` of the filtered model will translate between the two models, but in this case we map the indices using the appropriate visibility column. This handler assumes that an index indicating the view (*i*) is passed as user data.

```
moveSelected <- function(b, i) {
  selection <- selections[[i]]
  selected <- selection$getSelectedRows()
  if(length(selected$retval)) {
    childRows <- sapply(selected$retval, function(childPath) {
      childRow <- as.numeric(childPath$toString()) + 1L
    })
    shownIndices <- which(model[, 2L + i])
    rows <- shownIndices[childRows]

    model[rows, 2L + i] <- FALSE
  }
```

```

    model[rows, 2L + (3L-i)] <- !model[rows, 2L + i]
  }
}

```

We connect the handler to the "clicked" signal for the buttons.

```

mapapply(gSignalConnect, buttons, "clicked", list(moveSelected), 1:2)

```

```

fromLeft.clicked fromRight.clicked
588 589

```

We add one flourish, namely ensuring that the arrows are not sensitive when the corresponding selection is not set.

```

disableButton <- function(sel, button) {
  selected <- sel$getSelectedRows()
  button$setSensitive(length(selected$retval) != 0)
}
mapapply(gSignalConnect, selections, "changed", list(disableButton), buttons)

```

```

left.changed right.changed
590 591

```

As the initial state has no selection, we set the button sensitivities accordingly.

```

sapply(buttons, gtkWidgetSetSensitive, FALSE)

```

```

$fromLeft
NULL

```

```

$fromRight
NULL

```

Toggle cell renderers Binary data can be represented by a toggle. The `gtkCellRendererToggle` will create a check box in the cell that will appear checked if the active property is TRUE. If an attribute is defined for the property, then changes in the model will be reflected in the view. More work is required to modify the model in response to user interaction with the view. The `activatable` attribute for the cell must be TRUE in order for it to receive user input. The programmer needs to connect to the `toggled` to update the model in response to changes in the active state.

```

cr <- gtkCellRendererToggle()
cr['activatable'] <- TRUE # cell can be activated
cr['active'] <- TRUE
gSignalConnect(cr, "toggled", function(w, path) {
  model$active[as.numeric(path) + 1] <- w['active']
})

```

```
toggled
  593
attr(,"class")
[1] "CallbackID"
```

To render the toggle as a radio button instead of a check box, set the `radio` property to `TRUE`. The programmer is responsible for implementing the radio button logic via the `toggled` signal.

Example 8.4: Displaying a check box column in a tree view

This example demonstrates the construction of a GUI for selecting one or more rows from a data frame. We will display a list of the installed packages that can be upgraded from CRAN, although this code is trivially generalizable to any list of choices. The user selects a row by clicking on a checkbox produced by a toggle cell renderer.

To get the installed packages that can be upgraded, we use some of the functions provided by the `utils` package.

```
d <- old.packages()[,c("Package", "Installed", "ReposVer")]
d <- as.data.frame(d)
```

This function will be called on the selected rows. Here, we simply call `install.packages` to update the selected packages.

```
doUpdate <- function(d) {
  install.packages(d$Package)
}
```

To display the data frame, we first append a column to the data frame to store the selection information and then create a corresponding `RGtkDataFrame`.

```
n <- ncol(d)
nms <- colnames(d)
d$.toggle <- rep(FALSE, nrow(d))
store <- rGtkDataFrame(d)
```

Our tree view shows each text column using a simple text cell renderer, except for the first column that contains the check boxes for selection.

```
view <- gtkTreeView()
# add toggle
cr <- gtkCellRendererToggle()
view$insertColumnWithAttributes(0, "", cr, active = n)
```

```
[1] 1
```

```
cr['activatable'] <- TRUE
gSignalConnect(cr, "toggled", function(cr, path, user.data) {
  view <- user.data
```

```

row <- as.numeric(path) + 1
model <- view$getModel()
n <- dim(model)[2]
model[row, n] <- !model[row, n]
}, data=view)

```

```

toggled
  610
attr(,"class")
[1] "CallbackID"

```

The text columns are added one-by-one in a similar manner:

```

mapply(view$insertColumnWithAttributes, -1, nms, list(gtkCellRendererText()),
       text = 1:n-1)

```

```

[1] 2 3 4

```

Finally, we connect the store to the model.

```

view$setModel(store)

```

To allow the user to initiate the action, we create a button and assign a callback. We pass in the view, rather than the model, in case the model would be recreated by the `doUpdate` call. In a real application, once a package is upgraded it would be removed from the display.

```

b <- gtkButton("Update packages")
gSignalConnect(b, "clicked", function(w, data) {
  view <- data
  model <- view$getModel()
  n <- dim(model)[2]
  vals <- model[model[, n], -n, drop=FALSE]
  doUpdate(vals)
}, data=view)

```

```

clicked
  633
attr(,"class")
[1] "CallbackID"

```

Our basic GUI places the view into a box container that also holds the button to initiate the action.

```

w <- gtkWindow(show=FALSE)
w$setTitle("Installed packages that need upgrading")
w$setSizeRequest(300, 300)
g <- gtkVBox(); w$add(g)
sw <- gtkScrolledWindow()
g$packStart(sw, expand=TRUE, fill=TRUE)
sw$add(view)

```

```
sw$setPolicy("automatic", "automatic")
g$packStart(b, expand=FALSE)
w$show()
```

Rendering progress in cells To visually communicate progress within a cell, both progress bars and spinner animations are supported. These modes correspond to `GtkCellRendererProgress` and `GtkCellRendererSpinner`, respectively.

In the case of `GtkCellRendererProgress`, its value property takes a value between 0 and 100 indicating the amount finished, with a default value of 0. Values out of this range will be signaled by an error message. The orientation property, with values from `GtkProgressBarOrientation`, can adjust the direction that the bar grows. For example,

```
cr <- gtkCellRendererProgress()
cr["value"] <- 50 # fixed 50%
cr['orientation'] <- "right-to-left"
```

For indicating progress in the absence of a definite end point, `GtkCellRendererSpinner` is more appropriate. The spinner is displayed when the active property is TRUE. Increment the pulse property to drive the animation.

Overriding attribute mappings The default behavior for mapping model values to a renderer property is simple: values are extracted from the model and passed directly to the cell renderer property. If the data types are different, such as a numeric value for a string property, the value is converted using low-level routines defined by the property system. It is sometimes desirable to override this mapping with more complex logic.

For example, conversion of numbers to strings is a non-trivial task. Although the logic in the R print system often performs acceptably, there is certainly room for customization, for example aligning floating point numbers by fixing the number of decimal places. This could be done in the model (e.g., using `sprintf` to format and coerce to character data). However, performing the conversion during rendering requires one to intercept the model value before it is passed to the cell renderer. In the specific case `GtkTreeView`, it is possible to specify a callback that overrides this step.

The callback, of type `GtkTreeCellDataFunc`, accepts arguments for the tree view column, the cell renderer, the model, an iterator pointing to the row in the model and an argument for user data. The function is tasked with setting the appropriate attributes of the cell renderer. For example, this function could be used to format floating point numbers:

```
func <- function(viewCol, cellRend, model, iter, data) {
  curVal <- model$getValue(iter, 0)$value
  fVal <- sprintf("%.3f", curVal)
```



```
cellRend['text'] <- fVal
cellRend['xalign'] <- 1
}
```

The function then needs to be registered with a `GtkTreeViewColumn` that is rendering a numeric column from the model:

```
view <- gtkTreeView(rGtkDataFrame(data.frame(rnorm(100))))
cr <- gtkCellRendererText()
view$insertColumnWithAttributes(0, "numbers", cr, text = 0)
```

```
[1] 1
```

```
vc <- view$getColumn(0)
vc$setCellDataFunc(cr, func)
```

```
NULL
```

The last line is the key: calling `setCellDataFunc` registers our custom formatting function with the view column.

One drawback with the use of such functions is that R code is executed every time a cell is rendered. If performance matters, consider pre-converting the data in the model or tweaking the options in R for printing real numbers, namely `scipen` and `digits`.

For customizing rendering further and outside the scope of `GtkTreeView`, one could implement a new type of `GtkCellRenderer`. See Section ?? for more details on this advanced concept.

Editable cells When the `editable` property of a text cell (or `activatable` property of a toggle cell) is set to `TRUE`, then the cell contents can be changed. This allows the user to make changes to the underlying model through the GUI. Although the view automatically reflects changes made to the model, the reverse is not true. A callback must be assigned to the `editable` (`toggled`) signal for the cell renderer to implement the change. The callback for the "edited" signal has arguments `renderer`, `path` for the path of the selected row (as a string), and `new.text` containing the value of the edited text as a string. The tree view object and which column was edited are not passed in by default. These can be passed through the user data argument, set as user data on the renderer, or accessed from an enclosing environment if needed within the callback.

For example, here is how one can update an `RGtkDataFrame` model from within the callback.

```
cr['editable'] <- TRUE
ID <- gSignalConnect(cr, "edited",
  f=function(cr, path, newtext, user.data) {
    curRow <- as.numeric(path) + 1
```

```
curCol <- user.data$column
model <- user.data$model
model[curRow, curCol] <- newtext
}, data=list(model=store, column=1))
```

Moving the cursor Users may expect that once a cell is edited, the next cell is then set up to be edited. In order to do this, one must advance the cursor and activate editing of the next cell. For `GtkTreeView`, this is done through the `setCursor` method. The `path` argument takes a tree path instance, the `column` argument a tree view column object, and the flag `start.editing` indicates whether to initiate editing.

8.2 Display of hierarchical data

Although the `RGtkDataFrame` model is a convenient implementation of `GtkTreeModel`, it has its limitations. Primary among them is its lack of support for hierarchical data. GTK+ implements `GtkTreeModel` with `GtkListStore` and `GtkTreeStore`, which respectively store non-hierarchical and hierarchical tabular data. `GtkListStore` is a flat table, while `GtkTreeStore` organizes the table into a hierarchy. Here, we discuss `GtkTreeStore`.

Loading hierarchical data

A tree store is constructed using `gtkTreeStore`. The column types are specified through a character vector at the time of construction. The specification uses “GTypes” such as `gchararray` for character data, `gboolean` for logical data, `gint` for integer data, `gdouble` for numeric data, and `GObject` for GTK+ objects, such as `pixbufs`.

Example 8.5: Defining a tree

Below, we create a tree based on the `Cars93` dataset, where the car models are organized by manufacturer, i.e., each model row is the child of its manufacturer row:

```
tstore <- gtkTreeStore("gchararray")
by(Cars93, Cars93$Manufacturer, function(df) {
  piter <- tstore$append() # parent
  tstore$setValue(piter$iter, column = 0, value = df$Manufacturer[1])
  sapply(df$Model, function(model) {
    sibiter <- tstore$append(parent = piter$iter) # child
    if (is.null(sibiter$retval))
      tstore$setValue(sibiter$iter, column = 0, value = model)
  })
})
```

To retrieve a value from the tree store using its path we have:

```
iter <- tstore$getIterFromString("0:0")
tstore$getValue(iter$iter, column = 0)$value
```

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

This obtains the first model from the first manufacturer.

As shown in this example, populating a tree store relies on two functions: `append`, for appending rows, and `setValue`, for setting row values. The iterator to the parent row is passed to `append`. A parent of `NULL`, the default, indicates that the row should be at the top level. It would also be possible to insert rows using `insert`, `insertBefore`, or `insertAfter`. The `setValue` method expects the row iterator and the column index, 0-based.

An entire row can be assigned through the `set` method. The method uses positional arguments to specify the column and the value. The column index appears as an even argument (say $2k$) and the corresponding value in the odd argument (say $2k + 1$). Values are returned by the `getValue` method, in a list with component value storing the value.

Traversing a tree store is most easily achieved through the user of `GtkTreeIter`, introduced previously in the context of flat tables. Here we perform a depth-first traversal of our Cars93 model to obtain the model values:

```
## JV: had issues with this (older RGtk2?)
iter <- tstore$getIterFirst()
models <- NULL
while(iter$retval) {
  child <- tstore$iterChildren(iter$iter)
  while(child$retval) {
    models <- c(models, tstore$get(child$iter, 0)[[1]])
  }
  tstore$iterNext(iter$iter)
}
```

The hierarchical structure introduces the method `iterChildren` for obtaining an iterator to the first child of a row. As with other methods returning iterators, the return value is a list, with the `retval` component indicating the validity of the iterator, stored in the `iter` component. The method `iterParent` performs the reverse, iterating from child to parent.

Rows within a store can be rearranged using several methods. Call `swap` to swap rows referenced by their iterators. The methods `moveAfter` and `moveBefore` move one row after or before another, respectively. The `reorder` method totally reorders the rows under a specified parent given a vector of row indices, like that returned by `order`.

Once added, rows may be removed using the `remove` method. To remove every row, call the `clear` method.

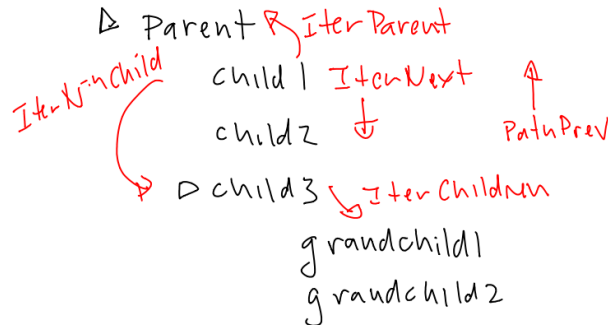


Figure 8.3: [REPLACEME!] Graphical illustration of the functions used by iterators to traverse a tree store.

Displaying data as a tree

Once a hierarchical dataset has been loaded into a `GtkTreeModel` implementation like `GtkTreeStore`, it can be passed to a `GtkTreeView` widget for display as a tree. Indeed, this is the same widget that displayed our flat data frame in the previous section. As before, `GtkTreeView` displays the `GtkTreeModel` as a table; however, it now adds controls for expanding and collapsing nodes where rows are nested.

The user can click to expand or collapse a part of the tree. These actions correspond to the signals `row-expanded` and `row-collapsed`, respectively.

Example 8.6: A simple tree display

Here, we demonstrate the application of `GtkTreeView` to the display of hierarchical data. We will use the model constructed in Example 8.5 from the `Cars93` dataset. In that example we defined a simple tree store from a data frame, with a level for manufacturer and make for different cars. We refer to that model by `tstore` below.

Now, we make a simple rectangular store for the make information with the following:

```
store <- rGtkDataFrame(Cars93[, "Model", drop=FALSE])
```

The basic view is similar to that for rectangular data already presented.

```
view <- gtkTreeView()
view$insertColumnWithAttributes(0, "Make", gtkCellRendererText(), text = 0)
```

```
[1] 1
```

Finally, we illustrate that the same view can be used with either model:

```
view$setModel(store)           # the rectangular store
view$setModel(tstore)          # or the tree store
```

Example 8.7: Dynamically growing a tree

This example uses a tree to explore an R list object, such as that returned by one of R's modelling functions. As the depth of these lists is not specified in advance, we use a dynamic approach to creating the tree store, modifying the tree store when the tree view is expanded or collapsed.

We begin by defining our tree store and an accompanying tree view. This example allows sorting, and so calls the `gtkTreeModelSort` function.

```
store <- gtkTreeStore(rep("gchararray", 2))
sstore <- gtkTreeModelSort(store)
```

We create a root row:

```
iter <- store$append(parent=NULL)$iter
store$setValue(iter, column=0, "GlobalEnv")
store$setValue(iter, column=1, "environment")
iter <- store$append(parent=iter)
```

It is necessary to append an empty row to the root so that root becomes expandable.

We now define the tree view and allow for multiple selection:

```
view <- gtkTreeView(sstore)
view$getSelection()$setMode("multiple")
```

The basic idea is to create child nodes when the parent is expanded and to delete the children when the parent is collapsed. This relies on the row-expanded and row-collapsed signals, respectively. First, we define the expansion handler:

```
gSignalConnect(view, signal = "row-expanded",
               f = function(view, iter, tpath, user.data) {
                 sortedModel <- view$getModel()
                 iter <- pathToIter(sortedModel, tpath)
                 path <- iterToRPath(sortedModel, iter)
                 children <- getChildren(path)
                 addChildren(store, children, parentIter=iter)
                 ## remove errant 1st offspring. See addChildren
                 ci <- store$iterChildren(iter)
                 if(ci$retval) store$remove(ci$iter)
               })
```

```
row-expanded
707
attr(,"class")
[1] "CallbackID"
```

The callback calls several helper functions to map the tree path to an R object, get the child components of the object and add them to the tree. The details are in the definitions of the helper functions.

The `pathToIter` function finds the iterator in the base tree model for a tree path in the sorted proxy.

```
pathToIter <- function(sstore, tpath) {  
  store <- sstore$getModel()  
  uspath <- sstore$convertPathToChildPath(tpath)  
  store$getIter(uspath)$iter  
}
```

We now need to convert the iterator to an “R path” is made up of the names of each component that makes up an element in the list. This function returns the path for a component specified by its iterator.

```
iterToRPath <- function(sstore, iter) {  
  store <- sstore$getModel()  
  indices <- store$getPath(iter)$getIndices()  
  iter <- NULL  
  path <- sapply(indices, function(i) {  
    iter <- store$iterNthChild(iter, i)$iter  
    store$getValue(iter, 0)$value  
  })  
  return(path[-1])  
}
```

The `getChildren` function obtains the child components of a given R object path. If the path is empty, the children are the objects in the global environment, the root. The return value is a `data.frame` with three columns: object name, object class and whether the object is recursive.

```
getChildren <- function(path=character(0)) {  
  hasChildren <- function(obj)  
    (is.list(obj) || is.environment(obj)) && !is.null(names(as.list(obj)))  
  
  getType <- function(obj) head(class(obj), n=1)  
  
  obj <- if (!length(path))  
    .GlobalEnv  
  else eval(parse(text=paste(path, collapse="$")), envir=.GlobalEnv)  
  
  children <- as.list(obj)  
  
  d <- data.frame(children = names(children),  
                  class = sapply(children, getType),  
                  offspring = sapply(children, hasChildren))  
  
  ## filter out Gtk ones  
  d[!grepl("^Gtk", d$class), ]  
}
```

The final step in the expansion handler is to add the children to the tree store with the `addChildren` function. Its one quirk is the addition of a dummy child row when the item has children. This makes the node expandable, i.e., the tree view draws an icon for the user to click to request the expansion.

```
addChildren <- function(store, children, parentIter = NULL) {
  if(nrow(children) == 0)
    return(NULL)
  for(i in 1:nrow(children)) {
    iter <- store$append(parent=parentIter)$iter
    sapply(1:(ncol(children) - 1), function(j)
      store$setValue(iter, column = j - 1, children[i, j]))
    ## Add a branch if there are children
    if(children[i, "offspring"])
      store$append(parent=iter)
  }
}
```

Next, we define a handler for the row-collapsed signal, which has a similar signature as the row-expanded signal. The handler removes the children of the newly collapsed node, so that we can add them again when the node is expanded.

```
gSignalConnect(view, signal = "row-collapsed",
  f = function(view, iter, tpath, user.data) {
    sortedModel <- view$getModel()
    iter <- pathToIter(sortedModel, tpath)
    n = store$iterNChildren(iter)
    if(n > 1) { ## n=1 gets removed when expanded
      for(i in 1:(n-1)) {
        child.iter <- store$iterChildren(iter)
        if(child.iter$retval)
          store$remove(child.iter$iter)
      }
    }
  })
```

```
row-collapsed
      708
attr(,"class")
[1] "CallbackID"
```

Our last handler simply demonstrates the retrieval of an object when its row is activated, i.e., double-clicked:

```
gSignalConnect(view, signal = "row-activated",
  f = function(view, tpath, tcol) {
    sortedModel <- view$getModel()
```

```
    iter <- pathToIter(sortedModel, tpath)
    path <- iterToRPath(sortedModel, iter)
    sel <- view$getSelection()
    out <- sel$getSelectedRows()
    if(length(out) == 0) return(c()) # nothing
    vals <- c()
    for(i in out$retval) { # multiple selections
      iter <- sortedModel$getIter(i)$iter
      newValue <- sortedModel$getValue(iter, 0)$value
      vals <- c(vals, newValue)
    }
    print(vals) # [Replace this]
  })
```

```
row-activated
      709
attr(,"class")
[1] "CallbackID"
```

To finish this example, we would need to populate the tree view with columns and display the view in a window.

8.3 Model-based combo boxes

Basic combo box usage was discussed in Section 7.5; here we discuss the more flexible and complex approach of using an explicit data model for storing the menu items. The item data is tabular, although it is limited to a single column. Thus, `GtkTreeModel` is again the appropriate model, and `RGtkDataFrame` is usually the implementation of choice.

To construct a `GtkComboBox` based on a user-created model, one should pass the model to the constructor `gtkComboBox`. This model may be changed or set through the `setModel` method and is returned by `getModel`. It remains to instruct the combo box how to display one or more data columns in the menu. Like `GtkTreeViewColumn`, `GtkComboBox` implements the `GtkCellLayout` interface and thus delegates the rendering of model values to `GtkCellRenderer` instances that are packed into the combo box.

As introduced in the previous chapter, the `GtkComboBoxEntry` widget extends `GtkComboBox` to provide an entry widget for the user to enter arbitrary values. To construct a combo box entry on top of a tree model, one should pass the model, as well as the column index that holds the textual item labels, to the `gtkComboBoxEntry` constructor. It is not necessary to create a cell renderer for displaying the text, as the entry depends on having text labels and thus enforces their display. It is still possible, of course, to add cell renderers for other model columns.

The `getActiveIter` returns a list containing the iterator pointing to the currently selected row in the model. If no row has been selected, the `retval`

component of the list is `FALSE`. The `setActiveIter` sets the currently selected item by iterator. As discussed previously, the `getActive` and `setActive` behave analogously with 0-based indices.

Signals When a user selects a value with the mouse, the `changed` signal is emitted. For combo box entry widgets, the `changed` signal will also be emitted when a new value has been entered. also make changes by typing in the new value. To detect when the user has finished entering text, one needs to retrieve the underlying `GtkEntry` widget with `getChild` and connect to its `activate` signal.

Example 8.8: Modifying the values in a combobox

This example employs two combo boxes: one to select a data frame and the other to select a variable from the selected data frame. This requires the variable selection combo box to update whenever the selected data frame changes. The data frame box will be a `GtkComboBoxEntry`, so that the user can enter names of objects in the current session. The other will be an ordinary combo box explicitly based on a `GtkTreeModel`, specifically an `RGtkDataFrame`. First, we construct the data frame box and populate it with some datasets that are provided with R:

```
datasets <- c("mtcars", "Cars93")
rdf <- rGtkDataFrame(datasets)
#dfCb <- gtkComboBoxEntryNewWithModel(rdf, text.column = 2)
dfCb <- gtkComboBoxEntry(); dfCb$model <- rdf
```

For the variable names, we construct a combo box based on an `RGtkDataFrame` and pack a text cell renderer to display the names:

```
variableNames <- character(0)
varModel <- rGtkDataFrame(variableNames)
varCb <- gtkComboBox(varModel)
cr <- gtkCellRendererText()
varCb$packStart(cr)
varCb$addAttribute(cr, "text", 0) # column 1
```

We now implement a signal handler that will update the variable combo box upon a selection or entry in the data frame box.

```
newDfSelected <- function(varCb, w, ...) {
  if(inherits(w, "GtkComboBox")) # get entry widget
    w <- w$getChild()
  val <- w$getText()
  df <- try(get(val, envir=.GlobalEnv), silent=TRUE)
  if(!inherits(df, "try-error") && is.data.frame(df)) {
    nms <- names(df)
    ## update model
    newModel <- rGtkDataFrame(nms)
  }
```

```
varCb$setModel(newModel)
varCb$setActive(-1)
}
}
gSignalConnect(dfCb, "changed", f=newDfSelected,
               user.data.first=TRUE,
               data=varCb)
```

```
changed
  795
attr(,"class")
[1] "CallbackID"
```

```
gSignalConnect(dfCb$getChild(), "activate", f=newDfSelected,
               user.data.first=TRUE,
               data=varCb)
```

```
activate
  797
attr(,"class")
[1] "CallbackID"
```

This callback is connected to the signals for both activation of the entry widget and the selection in the combo box. The function first checks whether it is responding to the combo box or the entry, and, if it is the combo box, it obtains the entry widget. To update the display we replace the model, which admittedly defeats much of the purpose of using a model.

Example 8.9: A color selection widget

This examples shows how a combobox can be used as an alternative to `gtkColorButton` to select a color. We use two cell renderers for each row, one to draw the color and the other a text label.

Our model has a single column: the color name from the R palette.

```
model <- rGtkDataFrame(palette())
```

Next we create the combo box with the model and add two cell renderers: one that fills itself with the color, the other that displays the text.

```
combobox <- gtkComboBox(model)
## color
crc <- gtkCellRendererText()
combobox$packStart(crc, expand=FALSE)
combobox$addAttribute(crc, "cell-background", 0)
crc$width <- 25
## text
crt <- gtkCellRendererText()
crt['xpad'] <- 5 # give some space
```

```
combobox$packStart(crt)
combobox$addAttribute(crt, "text", 0)
```

We use a `GtkCellRendererText` to display the color as its background, without any text, since we cannot create an instance of the abstract base class `GtkCellRenderer`.

8.4 Text entry widgets with completion

Often, the number of possible choices is too large to list in a combo box. One example is a web-based search engine: the possible search terms, while known and finite in number, are too numerous to list. The auto-completing text entry has emerged as an alternative to a combo box and might be described as a sort of dynamic combo box entry widget. When a user enters a string, partial matches to the string are displayed in a menu that drops down from the entry.

The `GtkEntryCompletion` object implements text completion in GTK+. An instance is constructed with `gtkEntryCompletion`. The underlying database is a `GtkTreeModel`, like `RGtkDataFrame`, set via the `setModel` method. To connect a `GtkEntryCompletion` to an actual `GtkEntry` widget, call the `setCompletion` method on `GtkEntry`. The `text-column` property specifies the column containing the completion candidates.

There are several properties that can be adjusted to tailor the completion feature; we mention some of them. Setting the property `inline-selection` to `TRUE` will place the top completion suggestion to the entry inline as the completions are scrolled through; `inline-completion` will add the common prefix automatically to the entry widget; `popup-single-match` is a logical indicating if a popup is displayed on a single match; `minimum-key-length` takes an integer specifying the number of characters needed in the entry before completion is checked (the default is 1).

By default, the rows in the data model that match the current value of the entry widget in a case insensitive manner are displayed. This matching function can be overridden by setting a new R function through the `setMatchFunc` method. The signature of this function is the completion object, the string from the entry widget (lower case), an iterator pointing to a row in the model and optionally user data that is passed through the `func.data` argument of the `setMatchFunc` method. This callback should return `TRUE` or `FALSE` depending on whether that row should be displayed in the set of completions.

Example 8.10: Text entry with completion

This example illustrates the steps to add completion to a text entry.

The two basic widgets are defined as follows:

```
entry <- gtkEntry()
```

```
completion <- gtkEntryCompletion()
entry$setCompletion(completion)
```

We will use an `RGtkDataFrame` instance for our completion model, taking a convenient list of names for our example. We set the model and text column index on the completion object and then set some properties to customize how the completion is handled.

```
store <- rGtkDataFrame(state.name)
completion$setModel(store)
completion$setTextColumn(0)
completion['inline-completion'] <- TRUE
completion['popup-single-match'] <- FALSE
```

We wish for the text search to match against any part of a string, not only the beginning. Thus, we need to define our own match function. We get the string from the entry widget, not the passed value, as that has been standardized to lower case.

```
matchAnywhere <- function(comp, str, iter, user.data) {
  model <- comp$getModel()
  rowVal <- model$getValue(iter, 0)$value # column 0 in model

  str <- comp$getEntry()$getText() # case sensitive
  grepl(str, rowVal)
}
completion$setMatchFunc(matchAnywhere)
```

```
NULL
```

8.5 Text views and text buffers

Multiline text areas are displayed through `GtkTextView` instances. These provide a view of an accompanying `GtkTextBuffer`, which is the model that stores the text and other objects to be rendered. The view is responsible for the display of the text in the buffer and has methods for adjusting tabs, margins, indenting, etc. The text buffer stores the actual text, and its methods are for adding and manipulating the text.

A text view is created with `gtkTextView`. The underlying text buffer can be passed to the constructor. Otherwise, a buffer is automatically created. This buffer is returned by the method `getBuffer` and may be set with the `setBuffer` method. Text views provide native scrolling support and thus are easily added to a scrolled window (Section 7.6).

Text may be added programmatically through various methods of the text buffer. The most basic `setText` which simply replaces the current text. The method `insertAtCursor` will add the text to the buffer at the current position of the cursor. Other means are described in the following sections.

GtkTextBuffer Details

In order to do more with a text buffer, such as retrieve the text, retrieve a selection, or modify attributes of just some of the text, one needs to become familiar with how pieces of the buffer are referred to within GTK+.

There are two methods: text iterators (iters) are a transient means to mark begin and end boundaries within a buffer, whereas text marks specify a location that remains when a buffer is modified. One can use these with tags to modify attributes of pieces of the buffer.

Iterators An *iterator* is a programming object used to traverse through some data, such as a text buffer or table of values. Iterators are typically transient. They have methods to indicate what they point to and often update these values without an explicit function call. Such behaviour is unusual for typical R programming.

In GTK+ a *text iterator* is used to specify a position in a buffer. Iterators become invalid as soon as a buffer changes, say through the addition of text. In RGtk2, iterators are stored as lists with components `iter` to hold a pointer to the underlying iterator and component `retval` to indicate whether the iterator when it was returned is valid. Many methods of the text buffer will update the iterator. This can happen inside a function call where the iterator is passed as an argument – basically a copy is not passed in. The copy method will create a copy of an iterator, in case one is to be modified but it is important to keep the original.

Several methods of the text buffer return iterators marking positions in the buffer. The beginning and end of the buffer are returned by the methods `getStartIter` and `getEndIter`. Both of these iters are returned at once by the method `getBounds` again as components of a list, in this case `start` and `end`. The current selection is returned by the method `getSelectionBounds`. Again, as a list of iterators specifying the start and end positions of the current selection. If there is no selection, then the component `retval` will be `FALSE`, otherwise it is `TRUE`.

The method `getIterAtLine` will return an iterator pointing to the start of the line, which is specified by 0-based line number. The method `getIterAtLineOffset` has an additional argument to specify the offset for a given line. An offset counts the number of individual characters and keeps track of the fact that the text encoding, UTF-8, may use more than one byte per character. In addition to the text buffer, a text view also has the method `getIterAtLocation` to return the iterator indicating the between-word space in the buffer closest to the point specified in *x-y* coordinates.

There are several methods for iterators that allow one to refer to positions in the buffer relative to the iterator, for example, these with obvious names to move a character or characters: `forwardChar`, `forwardChars`, `backwardChar`, and `backwardChars`. As well, there are methods to move to the end or be-

ginning of the word the iterator is in or the end or beginning of the sentence (`forwardWordEnd`, `backwardWordStart`, `backwardSentenceStart`, and `forwardSentenceEnd`). There are also various methods, such as `insideWord`, returning logical values indicating if the condition is met. To use these methods, the iterator in the `iter` component is used, not the value returned as a list. Example 8.11 shows how some of the above are used, in particular how these methods update the iterator rather than return a new one.

Modifying the buffer Iterators are specified as arguments to several methods to set and retrieve text. The `insert` method will insert text at a specified iterator. The argument `len` specifies how many bytes of the `text` argument are to be inserted. The default value of `-1` will insert the entire text. This method, by default, will also update the iterator to indicate the end of where the text is inserted. The `delete` method will delete the text between the iterators specified to the arguments `start` and `end`. The `getText` method will get the text between the specified `start` and `end` iterators. A similar method `getSlice` will also do this, only it includes offsets to indicate the presence of images and widgets in the text buffer.

Example 8.11: Finding the word one clicks on

This example shows how one can find the iterator corresponding to a mouse-button-press event. The callback has an event argument which is a `GdkEventButton` object with methods `getX` and `getY` to extract the `x` and `y` components of the event object. These give the position relative to the widget.¹

```
ID <- gSignalConnect(tv, "button-press-event", f=function(w, e, ...) {
  siter <- w$getIterAtLocation(e$getX(), e$getY())$iter
  niter <- siter$copy()                # need copy
  siter$backwardWordStart()
  niter$forwardWordEnd()
  val <- w$getBuffer()$getText(siter, niter)
  print(val)                          # replace
  return(FALSE)                      # call next handler
})
```

Marks In addition to iterators, GTK+ provides marks to indicate positions in the buffer that persist through changes. For instance, the mark `"insert"` always refers to the position of the cursor. Marks have a gravity of `"left"` or `"right"`, with `"right"` being the default. When the text surrounding a mark is deleted, if the gravity is `"right"` the mark will remain to the right of any added text.

¹The methods `getXRoot` and `getYRoot` give the position relative to the parent window the widget resides in.

Marks can be defined in two steps by calling `gtkTextMark`, specifying a name and a value for the gravity, and then positioned within a buffer, specified by an iterator, through the buffer's `addMark` method. The `createMark` method combines the two steps.

There are many text buffer methods to work with marks. The `getMark` method will return the mark object for a given name. (There are functions which refer to the name of a mark, and others requiring the mark object.) The method `getIterAtMark` will return an iterator for the given mark to be used when an iterator is needed.

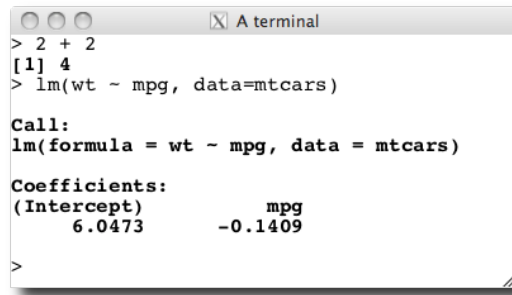
Tags Marks and iterators can be used to specify different properties for different parts of the text buffer. GTK+ uses tags to specify how pieces of text will differ from those of the textview overall. To create a tag, the `createTag` method is used. This has optional argument `tag.name` which can be used to refer to the tag later, and otherwise uses named arguments so specify a properties names and the corresponding values. These tags may be applied to the text between two iters using the methods `applyTag` or `applyTagByName`.

Example 8.12: Using text tags

We define two text tags to make text bold or italic and illustrate how to apply them.

```
tv <- gtkTextView()
tb <- tv$getBuffer()
tb$setText("the quick brown fox jumped over the lazy dog")
##
tag.b <- tb$createTag(tag.name="bold",
                      weight=PangoWeight["bold"])
tag.em <- tb$createTag(tag.name="em",
                      style=PangoStyle["italic"])
tag.large <- tb$createTag(tag.name="large",
                         font="Serif normal 18")
##
iter <- tb$getBounds()           # or get iters another way
tb$applyTag(tag.b, iter$start, iter$end) # updates iters
tb$applyTagByName("em", iter$start, iter$end)
```

Interacting with the clipboard GTK+ can create clipboards and provides convenient access to the default clipboard so that the standard cut, copy and paste actions can be implemented. The function `gtkClipboardGet` returns the default clipboard if given no arguments. The clipboard is the lone argument for the method `copyClipboard` to copy the current selection to the clipboard. The method `cutClipboard` has an extra argument, `default.editable`, which is typically `TRUE`. The `pasteClipboard` method is used paste the clipboard contents into the buffer, the second argument is `NULL` to paste at the



```
> 2 + 2
[1] 4
> lm(wt ~ mpg, data=mtcars)

Call:
lm(formula = wt ~ mpg, data = mtcars)

Coefficients:
(Intercept)      mpg
   6.0473      -0.1409
>
```

Figure 8.4: A basic R terminal implemented using a `gtkTextView` widget.

insert are, or an iterator specifying otherwise where the text should be inserted. The third argument is `TRUE` if the pasted text is to be editable.

Example 8.13: A simple command line interface

This example shows how the text view widget can be used to make a simple command line. While programming a command line is unlikely to be a common task in designing a GUI for a statistics application, the example is familiar and shows several different, but useful, aspects of the widget.

We begin by defining our text view widget and retrieving its buffer. We also specify a fixed-width font for the buffer.

```
tv <- gtkTextView()
tb <- tv$getBuffer()
font <- pangoFontDescriptionFromString("Monospace")
tv$modifyFont(font) # widget wide
```

We will use a few formatting tags, defined next. We don't need the tag objects, as we refer to them later by name.

```
aTag <- tb$createTag(tag.name="cmdInput")
aTag <- tb$createTag(tag.name="cmdOutput",
                    weight=PangoWeight["bold"])
aTag <- tb$createTag(tag.name="cmdError",
                    weight=PangoStyle["italic"], foreground="red")
aTag <- tb$createTag(tag.name="uneditable", editable=FALSE)
```

We define one new mark to indicate the prompt for a new line. We need to be able to identify a new command, and this marks its beginning.

```
startCmd <- gtkTextMark("startCmd", left.gravity=TRUE)
tb$addMark(startCmd, tb$getStartIter()$iter)
```

We define several helper functions. This first shows how to move the viewport to the end of the buffer so that the command line is visible.


```
scrollViewport <- function(view, ...) {  
  iter <- view$getBuffer()$getEndIter()$iter  
  view$scrollToIter(iter, 0)  
}
```

There are two types of prompts needed: one for entering a new command and one for a continuation. This function adds either, depending on its argument.

```
addPrompt <- function(obj, prompt=c("prompt", "continue"),  
                      setMark=TRUE)  
{  
  prompt <- match.arg(prompt)  
  prompt <- getOption(prompt)  
  
  endIter <- obj$getEndIter()  
  obj$insert(endIter$iter, prompt)  
  if(setMark)  
    obj$moveMarkByName("startCmd", endIter$iter)  
  obj$applyTagByName("uneditable", obj$getStartIter()$iter,  
                    obj$getEndIter()$iter)  
}  
addPrompt(tb) ## place an initial prompt
```

This helper method writes the output of a command to the text buffer. We arrange to truncate large outputs. By passing in the tag name, we can specify whether this is normal output or an error message.

```
addOutput <- function(obj, output, tagName="cmdOutput") {  
  endIter <- obj$getEndIter()  
  if(length(output) > 0)  
    sapply(output, function(i) {  
      obj$insertWithTagsByName(endIter$iter, i, tagName)  
      obj$insert(endIter$iter, "\n", len=-1)  
    })  
}
```

This next function uses the startCmd mark and the end of the buffer to extract the current command. Multi-line commands are parsed with a regular expression.

```
findCMD <- function(obj) {  
  endIter <- obj$getEndIter()  
  startIter <- obj$getIterAtMark(startCmd)  
  cmd <- obj$getText(startIter$iter, endIter$iter, TRUE)  
  regex <- paste("\n[", getOption("continue"), "] ", sep = "")  
  cmd <- unlist(strsplit(cmd, regex))  
  cmd  
}
```

The following function takes the current command and evaluates it. It uses a hack (involving `grep`) to distinguish between an incomplete command and a true syntax error.

```
evalCMD <- function(obj, cmd) {
  cmd <- paste(cmd, sep="\n")
  out <- try(parse(text=cmd), silent=TRUE)
  if(inherits(out, "try-error")) {
    if(length(grep("end", out))) {      # unexpected end of input
      ## continue
      addPrompt(obj, "continue", setMark=FALSE)
    } else {
      ## error
      addOutput(obj, out, tagName = "cmdError")
    }
    return()
  }

  e <- parse(text = cmd)
  out <- capture.output(vis <- withVisible(try(eval(e, .GlobalEnv), TRUE)))

  addOutput(obj, out)
  if (inherits(vis$value, "try-error"))
    addOutput(obj, vis$value, "cmdError")
  else if (vis$visible)
    addOutput(obj, capture.output(print(vis$value)))

  addPrompt(obj, "prompt", setMark=TRUE)
}
```

The `evalCMD` command is called when the return key is pressed. The `key-release-event` signal passes the event information through to the second argument. We inspect the key value and compare to that of the return key.

```
gSignalConnect(tv, "key-release-event", f=function(w, e, data) {
  obj <- w$getBuffer()          # w is textview
  keyval <- e$getKeyval()
  if(keyval == GDK_Return) {
    cmd <- findCMD(obj)          # character(0) if nothing
    if(length(cmd) && nchar(cmd) > 0)
      evalCMD(obj, cmd)
  }
  return(FALSE)                 # events need return value
})
```

```
key-release-event
          942
attr(,"class")
```

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

Finally, We connect `moveViewport` to the `changed` signal of the text buffer, so that the view always scrolls to the bottom when the contents of the buffer are modified:

```
gSignalConnect(tb, "changed", scrollViewport, tv, after = TRUE,  
               user.data.first = TRUE)
```

```
changed  
  943  
attr(,"class")  
[1] "CallbackID"
```

Figure 8.4 shows the widget placed into a very simple GUI.

Inserting non-text items If desired, one can insert images and/or widgets into a text buffer, although this isn't a common use within statistical GUIs. The method `insertPixbuf` will insert into a position specified by an iter a `GdkPixbuf` object. In the buffer, this will take up one character, but will not be returned by `getText`.

Arbitrary child components can also be inserted. To do so an anchor must first be created in the text buffer. The method `createChildAnchor` will return such an anchor, and then the text view method `addChildAtAnchor` can be used to add the child.

GtkTextView Details

Properties Key properties of the text view include `editable`, which if assigned a value of `FALSE` will prevent users from editing the text. If the view is not editable, the cursor may be hidden by setting the `cursor-visible` property to `FALSE`. The text in a view may be wrapped or not. The method `setWrapMode` takes values from `GtkWrapMode` with default of `"none"`, but options for `"char"`, `"word"`, or `"word_char"`. The justification for the entire buffer is controlled by the `justification` property which takes values of `"left"`, `"right"`, `"center"`, or `"fill"` from `GtkJustification`. The global value may be overridden for parts of the text buffer through the use of text tags. The left and right margins are adjusted through the `left-margin` and `right-margin` properties.

The text buffer has a few key properties, including `text` for storing the text and `has-selection` to indicate if text is currently selected in a view. The buffer also tracks if it has been modified. This information is available through the buffer's `getModified` method, which returns `TRUE` if the buffer has changes. The method `setModified`, if given a value of `FALSE`, allows the programmer to change this state, say after saving a buffer's contents.

Fonts The size and font can be globally set for a text view using the `modifyFont` method. (Specifying fonts for parts of the buffer requires the use of tags, described later.) The argument `font.desc` specifies the new font using a Pango font description, which may be generated from a string specifying the font through the function `pangoFontDescriptionFromString`. These strings may contain up to 3 parts: the first is a comma-separated list of font families, the second a white-space separated list of style options, and the third a size in points or pixels if the units “px” are included. A typical value might look like “serif, monospace bold italic condensed 16”. The various style options are enumerated in `PangoStyle`, `PangoVariant`, `PangoWeight`, `PangoStretch`, and `PangoGravity`. The help page for `PangoFontDescription` contains more information.

Signals The text buffer emits many different types of signals detailed in the help page for `gtkTextBuffer`. Most importantly, the `changed` signal is emitted when the content of the buffer changes. The callback for a `changed` signal has signature that returns the text buffer and any user data.

RGtk2: Menus and Dialogs

In the traditional WIMP-style GUI, the user executes commands by selecting items from a menu. In GUI terminology, such a command is known as an *action*. A GUI may provide more than one control for executing a particular action. Menubars and toolbars are the two most common widgets for performing application-wide actions. In this chapter, we will introduce actions, menus and toolbars and conclude by explaining the mechanisms in GTK+ for conveniently defining and managing actions and associated widgets in a large application.

9.1 Actions

GTK+ represents actions with the `GtkAction` class. A `GtkAction` can be proxied by widgets like buttons in a `GtkMenubar` or `GtkToolbar`. The `gtkAction` function is the constructor:

```
a <- gtkAction(name="ok", label="_Ok",
               tooltip="An OK button", stock.id="gtk-ok")
```

The constructor takes arguments `name` (to programmatically refer to the action), `label` (the displayed text), `tooltip`, and `stock.id` (identifying a stock icon). The command associated with an action is implemented by a callback connected to the `activate` signal:

```
gSignalConnect(a, "activate", f = function(w, data) {
  print(a$getName())           # or some useful thing
})
```

```
activate
  969
attr(,"class")
[1] "CallbackID"
```

An action plays the role of a data model describing a command, while widgets that implement the `GtkActivatable` interface are the views and controllers. All buttons, menu items and tool items implement `GtkActivatable`

and thus may serve as action proxies. Actions are connected to widgets through the method `connectProxy`:

```
b <- gtkButton()
a$connectProxy(b)
```

Certain aspects of a proxy widget are coordinated through the action. This includes sensitivity and visibility, corresponding to the `sensitive` and `visible` properties. To synchronize with aesthetic properties like the label and icon (e.g., `stock-id`), the `use-action-appearance` property must be set on the activatable widget:

```
## JV: this gave an error
a["use-action-appearance"] <- TRUE
```

Often, the commands in an application have a natural grouping. It can be convenient to coordinate the sensitivity and visibility of entire groups of actions. `GtkActionGroup` represents a group of actions. By convention, keyboard accelerators are organized by group, and the accelerator for an action is usually specified upon insertion:

```
group <- gtkActionGroup()
group$addActionWithAccel(a, "<control>O")
```

In addition to the properties already introduced, an action may have a shorter label for display in a toolbar (`short_label`), and hints for when to display its label (`is_important`) and image (`always_show_image`).

There is a special type of action that has a toggled state: `GtkToggleAction`. The `active` property represents the toggle. A further extension is `GtkRadioAction`, where the toggled state is shared across a list of radio actions, via the `group` property. Proxy widgets represent toggle and radio actions with controls resembling check boxes and radio buttons, respectively. Here, we create a toggle action for fullscreen mode:

```
fullScreen <- gtkToggleAction("fullscreen", "Full screen", "Toggle full screen")
gSignalConnect(fullScreen, "toggled", function(action) {
  window$fullscreen()
})
```

```
toggled
972
attr(,"class")
[1] "CallbackID"
```

We connect to the `toggled` signal to respond to a change in the action state.

9.2 Menus

A menu is a compact, hierarchically organized collection of buttons, each of which may proxy an action. Menus listing window-level actions are usually

contained within a menu bar at the top of the window or screen. Menus with options specific to a particular GUI element may “popup” when the user interacts with the element, such as by clicking the right mouse button. Menubars and popup menus may be constructed by appending each menu item and submenu separately, as illustrated below. For menus with more than a few items, we recommend the strategies described in Section 9.4.

Menubars

We will first demonstrate the menu bar, leaving the popup menu for later. The first step towards populating a menu bar is to construct the menu bar itself:

```
menubar <- gtkMenuBar()
```

A menu bar is a special type of container called a menu shell. An instance of `GtkMenuShell` contains one or more menu items. `GtkMenuItem` is an implementation of `GtkActivatable`, so each menu item can proxy an action. Usually, a menu bar consists multiple instances of the other type of menu shell: the menu, `GtkMenu`. Here, we create a menu object for our “File” menu:

```
fileMenu <- gtkMenu()
```

As a menu is not itself a menu item, we first must embed the menu into a menu item, which is labeled with the menu title:

```
fileItem <- gtkMenuItemNewWithMnemonic(label="_File")
fileItem$setSubmenu(fileMenu)
```

The underscore in the label indicates the key associated with the mnemonic for use when navigating the menu with a keyboard. Finally, we append the item containing the file menu to the menu bar:

```
menubar$append(fileItem)
```

In addition to append, it is also possible to prepend and insert menu items into a menu shell. As with any container, one can remove a child menu item, although the convention is to desensitize an item, through the `sensitive` property, when it is not currently relevant.

Next, we populate our file menu with menu items that perform some command. For example, we may desire an open item:

```
open <- gtkMenuItemNewWithMnemonic("_Open")
```

This item does not have an associated `GtkAction`, so we need to implement its activate signal directly:

```
gSignalConnect(open, "activate", function(item) {
  f <- file.choose()
  file.show(f)
})
```

```
activate
  986
attr(,"class")
[1] "CallbackID"
```

The item is now ready to be added to the file menu:

```
fileMenu$append(open)
```

It is recommended, however, to create menu items that proxy an action. This will facilitate, for example, adding an equivalent toolbar item later. First, we create the action:

```
saveAction <- gtkAction("save", "Save", "Save object", "gtk-save")
```

Then the appropriate menu item is generated from the action and added to the file menu:

```
save <- saveAction$createMenuItem()
##save["use-action-appearance"] <- TRUE
fileMenu$append(save)
```

Toggle menu items, i.e., a label next to a check box, are also supported. A toggle action will create one implicitly:

```
autoSaveAction <- gtkToggleAction("autosave", "Autosave", "Enable autosave")
autoSave <- autoSaveAction$createMenuItem()
fileMenu$append(autoSave)
```

A simple way to organize menu items, besides grouping into menus, is to insert separators between logical groups of items. Here, we insert a separator item, rendered as a line, to group the open and save commands a part from the rest of the menu:

```
fileMenu$append(gtkSeparatorMenuItem())
```

Popup Menu

Example 9.1: Popup menus

To illustrate popup menus, we show how construct one and display it in response to a mouse click. We start with a `gtkMenu` instance, to which we add some items.

```
popup <- gtkMenu() # top level
popup$append(gtkMenuItem("cut"))
popup$append(gtkMenuItem("copy"))
popup$append(gtkSeparatorMenuItem())
popup$append(gtkMenuItem("paste"))
```

Let us assume that we have a button that will popup a menu when clicked with the third (right) mouse button:


```
b <- gtkButton("Click me with right mouse button")
w <- gtkWindow(); w$setTitle("Popup menu example")
w$add(b)
```

This menu will be shown by calling `gtkMenuPopup` in response to the `button-press-event` signal on the button. The `gtkMenuPopup` function is called with the menu, some optional arguments for placement, and some values describing the event: the mouse button and time. The event values can be retrieved from the second argument of the callback (a `GdkEvent`).

```
gSignalConnect(b, "button-press-event",
  f = function(w, e, menu) {
    if(e$getButton() == 3 ||
      (e$getButton() == 1 && # a mac
        e$getState() == GdkModifierType['control-mask']))
      gtkMenuPopup(menu,
        button = e$getButton(),
        activate.time = e$getTime())
    return(FALSE)
  }, data=popup)
```

```
button-press-event
      1008
attr(,"class")
[1] "CallbackID"
```

The above will popup a menu, but until we bind to the `activate` signal, nothing will happen when a menu item is selected. Below we supply a stub for sake of illustration. The children of a popup menu are the menu items, including the separator which we avoid.

```
IDs <- sapply(popup$getChildren(), function(i) {
  if(!inherits(i, "GtkSeparatorMenuItem")) # skip these
    gSignalConnect(i, "activate",
      f = function(w, data) print("replace me"))
})
```

9.3 Toolbars

Toolbars are like menu bars in that they are containers for activatable items, but toolbars are not hierarchical. Thus, toolbars are not appropriate for storing a large number of items, only those that are activated most often.

We begin by constructing an instance of `GtkToolbar`:

```
toolbar <- gtkToolbar()
```

In analogous fashion to the menu bar, toolbars are containers for tool items. Technically, an instance of `GtkToolItem` could contain any type of widget, though toolbars typically represent actions with buttons. The `GtkToolButton`

class implements this common case. Here, we create a tool button for opening a file:

```
## openButton <- gtkToolButton(stock.id = "gtk-open")
## JV: had to do this:
openButton <- gtkToolButton()
openButton['stock-id'] <- "gtk-open"
```

Tool buttons have a number of properties, including label and several for specifying an icon. Above, we specify a stock identifier, for which there is a predefined translated label and theme-specific icon. As with any other container, the button may be added to the toolbar with the add method:

```
toolbar$add(openButton)
```

This appends the open button to the end of the toolbar. To insert into a specific position, we would call the insert method.

Usually, any application with a toolbar also has a menu bar, in which case many actions are shared between the two containers. Thus, it is often beneficial to construct a tool button directly from its corresponding action:

```
saveButton <- saveAction$createToolItem()
toolbar$add(saveButton)
```

A tool button is created for saveAction, created in the previous section.

Like menus, related buttons may be grouped using separators:

```
toolbar$add(gtkSeparatorToolItem())
```

Any toggle action will create a toggle tool button as its proxy:

```
fullScreenButton <- fullScreen$createToolItem()
toolbar$add(fullScreenButton)
```

A GtkToggleToolButton embeds a GtkToggleButton, which is depressed whenever its active property is TRUE.

A toolbar may be customized in a number of ways.

It is now our responsibility to place the toolbar at the top of the window, under the menu created in the previous section:

Example 9.2: Color menu tool button

Space in a toolbar is limited, and sometimes there are several actions that differ only by a single parameter. A good example is the color tool button found in many word processors. Including a button for every color in the palette would consume an excessive amount of space. A common idiom is to embed a drop-down menu next to the button, much like a combo box, for specifying the color, or, in general, any discrete parameter.

We demonstrate how one might construct a color-selecting tool button. Our menu will list the colors in the R palette. The associated button is a GtkColorButton. When the user clicks on the button, a more complex color selection dialog will appear, allowing total customization.

```
## JV: moved )
##gdkColor <- gdkColorParse(palette()[1]$color)
gdkColor <- gdkColorParse(palette()[1])$color
colorButton <- gtkColorButton(gdkColor)
```

gtkColorButton requires the initial color to be specified as a GdkColor, which we parse from the R color name.

The next step is to build the menu. Each menu item will display a 20x20 rectangle, filled with the color, next to the color name:

```
colorMenuItem <- function(color) {
  da <- gtkDrawingArea()
  da$setSizeRequest(20, 20)
  da$modifyBg("normal", color)
  item <- gtkImageMenuItem(color)
  item$setImage(da)
  item
}
colorItems <- sapply(palette(), colorMenuItem)
colorMenu <- gtkMenu()
for (item in colorItems)
  ## JV: fixed this
  ## menu$append(item)
  colorMenu$append(item)
```

An important realization is that the image in a GtkImageMenuItem may be any widget that presumably draws an icon; it need not be an actual GtkImage. In this case, we use a drawing area with its background set to the color. When an item is selected, its color will be set on the color button:

```
colorMenuItemActivated <- function(item) {
  color <- gdkColorParse(item$getLabel())$color
  colorButton$setColor(color)
}
sapply(colorItems, gSignalConnect, "activate", colorMenuItemActivated)
```

black.activate	red.activate	green3.activate	blue.activate
1072	1073	1074	1075
cyan.activate	magenta.activate	yellow.activate	gray.activate
1076	1077	1078	1079

Finally, we place the color button and menu together in the menu tool button:

```
menuButton <- gtkMenuToolButton(colorButton, "Color")
menuButton$setMenu(colorMenu)
toolbar$add(menuButton)
```

Toolbar items have some common properties. The buttons are comprised of an icon and text, and the style of their layout is specified by the toolbar

method `setStyle`, with values coming from the `GtkToolbarStyle` enumeration. Toolbar items can have a tooltip set for them through the methods `setTooltipText` or `setTooltipMarkup`, the latter if PANGO markup is desired. Toolbar items can be disabled, through the method `setSensitive`.

The items can be one of a few different types. A stock toolbar item is constructed by `gtkToolbarButtonNewFromStock`, with the stock id as the argument. The constructor `gtkToolbarButton` creates a button that can have its label and icon value set through methods `setLabel` and `setIconWidget`. Additionally, there are methods for setting a tooltip or specifying a stock id after construction. A toggle button, which toggles between looking depressed or not when clicked is created by `gtkToggleToolButton` (or `gtkToggleToolButtonNewFromStock`). Additionally there are constructors to place menus (`gtkMenuToolButton`) and radio groups (`gtkRadioToolButton`).

The clicked signal is emitted when a toolbar button is pressed. For the toggle button, the `toggle` signal is emitted.

Example 9.3: Basic toolbar usage

We illustrate with a toolbar whose buttons are produced in various ways.

```
tb <- gtkToolbar()
```

A button with a stock icon is produced by a call to the appropriate constructor.

```
b1 <- gtkToolButtonNewFromStock("gtk-open")
tb$add(b1)
```

To use a custom icons, requires a few steps.

```
f <- system.file("images/dataframe.gif", package="gWidgets")
image <- gtkImageNewFromFile(f)
b2 <- gtkToolButton()
b2$setIconWidget(image)
b2$setLabel("Edit")
tb$add(b2)
```

Adding a toggle button also is just a matter of calling the appropriate constructor. In this, example we illustrate how to initiate the callback only when the button is depressed.

```
b3 <- gtkToggleToolButtonNewFromStock("gtk-fullscreen")
tb$add(b3)
QT <- gSignalConnect(b3, "toggled", f=function(button, data) {
  if(button$getActive())
    cat("toggle button is depressed\n")
})
```

We give the other buttons a simple callback when clicked:

```
QT <- sapply(1:2, function(i) {
```

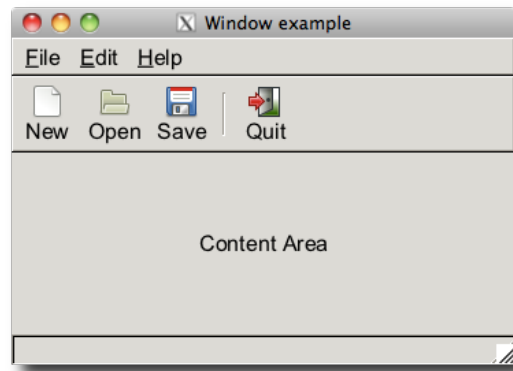


Figure 9.1: A GUI made using a UI manager to layout the menubar and toolbar.

```
gSignalConnect(tb[[i]], "clicked", function(button, data) {  
    cat("You clicked", button$getName(), "\n")  
})  
})
```

9.4 Managing a large user interface

A GUI is designed around actions that are accessible through the menubar and the toolbar. The notion of a *user interface manager* (UI manager) separates out the definitions of the actions from the user interface. The steps required to use GTK+'s UI manager are

1. define a UI manager,
2. set up an accelerator group for keyboard shortcuts,
3. define our actions,
4. create action groups to specify the name, label (with possible mnemonic), keyboard accelerator, tooltip, icon and callback for the graphical elements that call the action,
5. specify where the menu items and toolbar items will be placed,
6. connect the action group to the UI manager, and finally
7. display the widgets.

We show by an example how this is done.

Example 9.4: UI Manager example

We define the UI manager as follows

```
uimanager = gtkUIManager()
```

Our actions either open a dialog to gather more information or issue a command. A `GtkAction` element is passed to the action. We define a stub here, that simply updates a `gtkStatusbar` instance, defined below.

```
someAction <- function(action,...)
  statusbar$push(statusbar$getContextId("message"), action$getName())
Quit <- function(...) win$destroy()
```

To show how we can sequentially add interfaces, we break up our action group definitions into one for “File” and “Edit” and another one for “Help.” The key is the list defining the entries. Each component specifies (in this order) the name; the icon; the label, with `_` specifying the mnemonic; the keyboard accelerator, with `<control>`, `<alt>`, `<shift>` as possible prefixes, a tooltip, which may not work with the R event loop, and finally the callback. Empty values can be defined as `NULL` or, except for the callback, an empty string.

```
firstActionGroup = gtkActionGroup("firstActionGroup")
firstActionEntries = list(
  ## name,ID,label,accelerator,tooltip,callback
  file = list("File",NULL,"_File",NULL,NULL,NULL),
  new = list("New", "gtk-new", "_New", "<control>N",
    "New document", someAction),
  sub = list("Submenu", NULL, "S_ub", NULL, NULL, NULL),
  open = list("Open", "gtk-open", "_Open", "<ctrl>O",
    "Open document", someAction),
  save = list("Save", "gtk-save", "_Save", "<alt>S",
    "Save document", someAction),
  quit = list("Quit", "gtk-quit", "_Quit", "<ctrl>Q",
    "Quit", Quit),
  edit = list("Edit", NULL, "_Edit", NULL, NULL, NULL),
  undo = list("Undo", "gtk-undo", "_Undo", "<ctrl>Z",
    "Undo change", someAction),
  redo = list("Redo", "gtk-redo", "_Redo", "<ctrl>U",
    "Redo change", someAction)
)
```

We now add the actions to the action group, then add this action group to the first spot in the UI manager.

```
QT <- firstActionGroup$addActions(firstActionEntries)
uimanager$insertActionGroup(firstActionGroup, 0) # 0-based
```

The “Help” actions we do a bit differently. We define a “Use tooltips” mode to be a toggle, as an illustration of that feature. One can also incorporate radio groups, although this is not shown.

```
helpActionGroup = gtkActionGroup("helpActionGroup")
helpActionEntries = list(
  help = list("Help", "", "_Help", "", "", NULL),
```

```

    about = list("About", "gtk-about", "_About", "", "", someAction)
  )
  QT <- helpActionGroup$AddActions(helpActionEntries)

```

A toggle is defined with `gtkToggleAction` which has signature in a different order than the action entry. Notice, we don't have an icon, as the toggled icons is used. To add a callback, we connect to the `toggled` signal of the action element. This callback allows for user data, as illustrated.

```

toggleAction <- gtkToggleAction("UseTooltips",
                                label="_Use tooltips",
                                tooltip="Use tooltips ")
toggleAction$setActive(TRUE) # initially set
ID <- gSignalConnect(toggleAction, signal = "toggled",
                    f=function(ta, userData) {
                        cat(userData,ta$getName(),"\n")
                    },
                    data="toggled")
helpActionGroup$addAction(toggleAction)

```

We insert the help action group in position 2.

```

uimanager$insertActionGroup(helpActionGroup,1)

```

The `SetActive` method can set the state, use `GetActive` to retrieve the state.

Our UI Manager's layout is specified in a file. The file uses XML to specify where objects go. The structure of the file can be grasped quickly from the example. Each entry is wrapped in `ui` tags. The type of UI is either a menubar, toolbar, or popup. The name properties are used to reference the widgets later on. Menuitems are added with a `menuitem` entry and toolbar items the `toolitem` entry. These have an action value and an optional name (defaulting to the action value). The `separator` tags allow for some formatting. The nesting of the menuitems is achieved using the `menu` tags. A placeholder tag can be used to add entries at a later time.

```

<ui>
  <menubar name="menubar">
    <menu name="FileMenu" action="File">
      <menuitem name="FileNew" action="New"/>
      <menu action="Submenu">
<menuitem name="FileOpen" action="Open" />
      </menu>
      <menuitem name="FileSave" action="Save"/>
      <separator />
      <menuitem name="FileQuit" action="Quit"/>
    </menu>
    <menu action="Edit">

```

```
<menuitem name="EditUndo" action="Undo" />
<menuitem name="EditRedo" action="Redo" />
</menu>
<menu action="Help">
  <menuitem action="UseTooltips"/>
  <menuitem action="About"/>
</menu>
</menubar>
<toolbar name="toolbar">
  <toolitem action="New"/>
  <toolitem action="Open"/>
  <toolitem action="Save"/>
  <separator />
  <toolitem action="Quit"/>
</toolbar>
</ui>
```

This file is loaded into the UI manager as follows

```
id <- uimanager$addUiFromFile("ex-menus.xml")
```

The id value can be used to merge and delete UI components, but this is not illustrated here. The menus can also be loaded from strings.

Now we can setup a basic window template with menubar, toolbar, and status bar. We first get the three main widgets. We use the names from the UI layout to get the widgets through the `GetWidget` method of the UI manager. The menubar and toolbar are returned as follows, for our choice of names in the XML file.

```
menubar <- uimanager$getWidget("/menubar")
toolbar <- uimanager$getWidget("/toolbar")
```

The statusbar is constructed with

```
statusbar <- gtkStatusbar()
```

– in the definition of the callback `f` – is used to add new text to the statusbar. The `pop` method reverts to the previous message.

Now we define a top-level window and attach a keyboard accelerator group to the window so that when the window has the focus, the specified keyboard shortcuts can be used.

```
win <- gtkWindow(show=TRUE)
win$setTitle("Window example")
accelgroup = uimanager$getAccelGroup() # add accel group
win$addAccelGroup(accelgroup)
```

Now it is a simple matter of packing the widgets into a box.


```

box <- gtkVBox()
win$add(box)
box$packStart(menuBar, expand=FALSE, fill=FALSE, 0)
box$packStart(toolBar, expand=FALSE, fill=FALSE, 0)
contentArea = gtkVBox()
box$packStart(contentArea, expand=TRUE, fill=TRUE, 0)
contentArea$packStart(gtkLabel("Content Area"))
box$packStart(statusBar, expand=FALSE, fill=FALSE, 0)

```

The redo feature should only be sensitive to mouse events after a user has undone an action. To set the sensitivity of a menu item is done through the `SetSensitive` method called on the widget. We again retrieve the menu item or toolbar item widgets through their names.

```
uimanager$getWidget("/menubar/Edit/EditRedo")$setSensitive(FALSE)
```

To re-enable, use `TRUE` for the argument to `setSensitive`

We can also use the `SetText` method on the menu items. For instance, instead of a generic “Undo” label, one might want to change the text to list the most previous action. The method is not for the menu item though, but rather a `gtkLabel` which is the first child. We use the list notation to access that.

```

a <- uimanager$getWidget("/menubar/Edit/EditUndo")
a[[1]]$setText("Undo add text")

```

9.5 Dialogs

GTK+ comes with a variety of dialogs to create simple, usually single purpose, popup windows for the user to interact with.

The `gtkDialog` constructor

The constructor `gtkDialog` creates a basic dialog box, which is a display containing a top section with optionally an icon, a message, and a secondary message. The bottom section, the action area, shows buttons, such as yes, no and/or cancel. The convenience functions `gtkDialogNewWithButtons` and `gtkMessageDialog` simplify the construction.

In GTK+ dialogs can be modal or not. There are a few ways to make a dialog modal. The window method `setModal` will do so, as will passing in a modal flag to some of the constructors. These make other GUI elements inactive, but not the R session. Whereas, calling the `run` method, will stop the flow until the dialog is dismissed. The return value can then be inspected for the action, such as what button was pressed. These values are from `GtkResponseType`, which lists what can happen.

Basic message dialogs The `gtkMessageDialog` has an argument `parent`, to specify a parent window the dialog should appear relative to. The `flags` argument allows one to specify values (from `GtkDialogFlags`) of `destroy-with-parent` or `modal`. The `type` is used to specify the message type, using a value in `GtkMessageType`. The `buttons` is used to specify which buttons will be drawn. The `message` is the following argument. The dialog has a `secondary-text` property that can be set to give a secondary message.

```
w <- gtkWindow()
w['title'] <- "Parent window"
dlg <- gtkMessageDialog(parent=w, flags="destroy-with-parent",
                        type="question", buttons="ok",
                        "My message")
dlg['secondary-text'] <- "A secondary message"
response <- dlg$run()
if(response == GtkResponseType["cancel"] || # for other buttons
    response == GtkResponseType["close"] ||
    response == GtkResponseType["delete-event"]) {
  ## pass
} else if(response == GtkResponseType["ok"]) {
  print("Ok")
}
dlg$Destroy()
```

Making your own dialogs The `gtkDialog` constructor returns a dialog object which can be customized for more involved dialogs. In the example below, we illustrate how to make a dialog to accept user input. We use the `gtkDialogNewWithButtons`, which allows us to specify a stock button and a response value. We use standard responses, but could have used custom ones by specifying a positive integer. The dialog is a window object containing a box container, which is returned by the `getVbox` method. This box has a separator and button box packed in at the end, we pack in another box at the beginning below to hold a label and our entry widget.

When one of the buttons is clicked, the response signal is emitted by the dialog. We connect to this close the dialog.

```
dlg <- gtkDialogNewWithButtons(title="Enter a value",
                              parent=NULL, flags=0,
                              "gtk-ok", GtkResponseType["ok"],
                              "gtk-cancel", GtkResponseType["cancel"],
                              show=FALSE)
g <- dlg$getVbox() # content area
vg <- gtkVBox()
vg['spacing'] <- 10
g$packStart(vg)
vg$packStart(gtkLabel("Enter a value"))
```

```

entry <- gtkEntry()
vg$packStart(entry)
ID <- gSignalConnect(dlg, "response",
                    f=function(dlg, resp, user.data) {
                        if(resp == GtkResponseType["ok"])
                            print(entry$getText())
                        dlg$Destroy()
                    })
dlg$showAll()
dlg$setModal(TRUE)

```

File chooser

GTK+ has a `GtkFileChooser` backend to implement selecting a file from the file system. The same widget allows one to open or save a file and select or create a folder (directory). The action is specified through one of the `GtkFileChooserAction` flags. This backend presented in various ways through `gtkFileChooserDialog`, which pops up a modal dialog; `gtkFileChooserButton`, which pops up the dialog when the button is clicked; and `gtkFileChooserWidget`, which creates a widget that can be placed in a GUI to select a file.

The dialog constructor allows one to specify a title, a parent and an action. In addition, the dialog buttons must be specified, as with the last example using `gtkDialogNewWithButtons`.

Example 9.5: An open file dialog

An open file dialog can be created with:

```

dlg <- gtkFileChooserDialog(title="Open a file",
                           parent=NULL, action="open",
                           "gtk-ok", GtkResponseType["ok"],
                           "gtk-cancel", GtkResponseType["cancel"])

```

One can use the `run` method to make this modal or connect to the response signal. The file selected is found from the file chooser method `getFilename`. One can enable multiple selections, by passing `setSelectMultiple` a `TRUE` value. In this case, the `getFilenames` returns a list of filenames,

```

ID <- gSignalConnect(dlg, "response", f=function(dlg, resp, data) {
    if(resp == GtkResponseType["ok"]) {
        filename <- dlg$getFilename()
        print(filename)
    }
    dlg$destroy()
})

```

For the open dialog, one may wish to specify one or more filters, to narrow the available files for selection. A filter object is returned by the

`gtkFileFilter` function. This object is added to the file chooser, through its `addFilter` method. The filter has a name property set through the `setName` method. The user can select a filter through a combobox, and this provides the label. To use the filter, one can add a pattern (`addPattern`), a MIME type (`addMimeType`), or a custom filter.

```
fileFilter <- gtkFileFilter()
fileFilter$setName("R files")
dlg$addFilter(fileFilter)
QT <- sapply(c("*.R", "*.Rdata"),
             function(i) fileFilter$addPattern(i))
QT <- sapply(c("text/plain"),
             function(i) fileFilter$addMimeType(i))
```

The save file dialog is similar. The `setFilename` can be used to specify a default file and `setFolder` can specify an initial directory. To be careful as to not overwrite an existing file, the method `setDoOverwriteConfirmation` can be passed a `TRUE` value.

Date picker

A calendar widget is produced by `gtkCalendar`. This widget allows selection of a day, month or year. To specify these values, the properties `day`, `month` (0-11), and `year` store these values as integers. One can assign to these directly, or use the methods `selectDay` and `selectMonth` (no select year method). The method `getData` returns a list with components for the year, month and day. If there is no selection, the day component is 0.

The widget emits various signals when a selection is changed. The `day-selected` and `day-selected-double-click` ones are likely the most useful of these.

Programming GUIs using Qt

10.1 An introductory example

To get a small feel for how one programs a GUI using `qtbases`, the R package that interfaces R with the Qt libraries, we show how to produce a simple dialog to collect a date from a user.

If the underlying libraries and package are installed, the package is loaded as any other R package:

```
require(qtbases)
```

Constructors As with all other toolkits, in QtGUI components are created with constructors. For this example, we will set various properties later, rather than at construction time. For our GUI we have four basic widgets: a widget used as a container to hold the others, a label, a single-line edit area and a button.

```
w <- Qt$QWidget()
l <- Qt$QLabel()
e <- Qt$QLineEdit()
b <- Qt$QPushButton()
```

The constructors are not found in the global environment, but rather are found in the Qt environment provided through `qtbases`. As such, the `$` lookup operator is used. For this example, we use a `QWidget` as a top-level window, leaving for Section 11.6 to discuss the `QMainWindow` object and its task-tailored features.

Widgets in Qt have various properties that set the state of the object. For example, the widget object, `w`, has the `windowTitle` property that is adjusted as follows:

```
w$windowTitle <- "An example"
```

Qt objects are essentially environments. In the above, the named component `windowTitle` of the environment holds the value of the `windowTitle` property of the object, so the `$` use is simply that for environments.

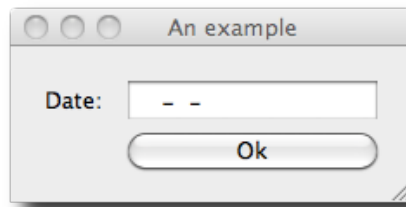


Figure 10.1: Screenshot of our sample GUI to collect a date from the user.

A more typical use is a method call. Qt overloads the `$` operator for method calls (as does RGtk2). For example, both the button object and label object have a text property. The setter `setText` can be used to assign a value. For example,

```
l$setText("Date:")
b$setText("Ok")
```

Although, the calling mechanism is more complicated than just the lookup of a function stored as the component `setText`, as the object is passed into the body of the function, the usage is similar.

Layout Managers Qt uses layout managers to organize widgets. This is similar to Java/Swing and `tcltk`, but not RGtk2. Layout managers will be discussed more thoroughly in Chapter 11, but in this example we will use a grid layout to organize our widgets. The placement of child widgets into the grid is done through the `addWidget` method and requires a specification, by index and span, of the cells the child will occupy.

```
lyt <- Qt$QGridLayout()
lyt$addWidget(l, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(e, 0, 1, 1, 1)
lyt$addWidget(b, 1, 1, 1, 1)
```

One can adjust properties of the layout, but we leave that discussion for later.

We need to attach our layout to the widget `w`, which is done through the `setLayout` method:

```
w$setLayout(lyt)
```

Then to view our GUI (Figure 10.1), we call the widget's `show` method.

```
w$show()
```

Callbacks As with other GUI toolkits, we add interactivity to our GUI by binding callbacks to certain events. To add a command to the clicking or

pressing of the button is done by attaching a handler to the “pressed” signal for the button (the “clicked” signal is only for mouse clicks). Widgets have various signals they emit. Additionally, there are window-manager events that may be of interest, but using them requires more work than is shown below. The `qconnect` function is used to add a handler for a signal. The function needs, at a minimum, the object, the signal name and the handler. Here we print the value stored in the “Date” field.

```
handler <- function(checked) print(e$text)
id <- qconnect(b, "pressed", handler)
```

We will discuss callbacks more completely in Section 10.6.

Refinements At this point, we have a working dialog built with `qtbase`. There is much room for refinement, which due to Qt’s many features are relatively easy to implement. For this example, we want to guide the user to fill out the date in the proper format. We could have used Qt’s `QDateEdit` widget to allow point-and-click selection, but instead show two ways to help the user fill in the information with the keyboard

The `QLineEdit` widget has a number of ways to adjust its behavior. For example, an input mask provides a pattern for the user to fill out. For a date, we may want the value to be in the form “year-month-date.” This would be specified with “0000-00-00”, as seen by consulting the API for `QLineEdit`. To add an input mask we have:

```
e$setInputMask("0000-00-00")
```

Further, for the line edit widget in Qt we can easily implement validation of the entered text. There are a few built-in validators, and for this purpose the regular expression validator could be used, but instead we wish to determine if we have a valid date by seeing if we can coerce the string value to a date via R’s `as.Date` function with a format of “%Y-%m-%d”. The method `setValidator` can be used to set the validator that is in charge of the validation. However, rather than passing a function, one must pass an instance of a validator class. For our specific needs, we need to create a new class.

Object-oriented support The underlying Qt libraries are written in C++. The object oriented nature is preserved by `qtbase`. Not only are Qt’s classes and methods implemented in R, the ability to implement new subclasses and methods is also possible. For the validation task, we need to implement a subclass of the `QValidator` class, and for this subclass implement a `validate` method. More detail on working with classes and methods in `qtbase` is provided in Section 10.8.

The `qsetClass` function is used to set a new class. To derive a subclass, we need just this:

```
qsetClass("dateValidator", Qt$QValidator)
```

The `validate` method is implemented as a virtual class in Qt, in R we implement it as a method of our subclass using the `qsetMethod` function. The signature of the `validate` method is a string containing the input and an index indicating where the cursor is in the text box. The return value of this method indicates a state of “Acceptable”, “Invalid”, or if neither can be determined “Intermediate.” Qt uses enumerations (cf. Section 10.7) to specify these values. The actual values are integers, but are kept as components of the environments provided by `qtbaseso` so can be referenced by name. In this case, the enumeration is in the `Qt$QValidator` class.

```
## FIXME: should probably use an input mask to restrict the input to
## 0000-00-00, then use as.Date() here to validate the date.
qsetMethod("validate", dateValidator, function(input, pos) {
  if(!grepl("[0-9]{4}-[0-9]{1,2}-[0-9]{1,2}$", input))
    return(Qt$QValidator$Intermediate)
  else if(is.na(as.Date(input, format="%Y-%m-%d")))
    return(Qt$QValidator$Invalid)
  else
    return(Qt$QValidator$Acceptable)
})
```

To use this new class, we call its constructor, which has the same name as the class, and then set it as a validator for the line edit widget:

```
validator <- dateValidator()
e$setValidator(validator)
```

10.2 The Qt library

Qt is an open-sourced, cross-platform application framework that is perhaps best known for its widget toolkit. The features of Qt are divided into about a dozen modules. We highlight some of the more important and interesting ones:

Core Basic utilities, collections, threads, I/O, ...

Gui Widgets, models, etc for graphical user interfaces

OpenGL Convenience layer (e.g., 2D drawing API) over OpenGL

Webkit Embeddable HTML renderer (shared with Safari, Chrome)

Other modules include functionality for networking, XML, SQL databases, SVG, and multimedia. However, R packages already provide many of those features.

The history of Qt begins with Haavard Nord and Eirik Chambe-Eng in 1991, and follows with the Trolltech company until 2008, and now Nokia, a major cell-phone producer. While it was originally not available as open-source on every platform, as of version 4 it was released universally under the GPL. With the release of Qt 4.5, Nokia placed Qt under the LGPL, so it is available for use in proprietary software, as well. Popular software developed

with Qt include the communication application Skype and the KDE desktop for Linux.

Qt is developed in C++ with extensions that require a special preprocessor called the *Meta Object Compiler* (MOC). The MOC allows for convenient syntax in the definition of signals, slots (signal handlers), and properties, which behave very similarly to those of GTK+.

There are many languages with bindings to Qt, and R is one such language. The `qtbases` package interfaces with every module of Qt. As its name suggests, `qtbases` forms the base for a number of R packages that provide high-level special-purpose interfaces to Qt. As these are still early in development, they will not be mentioned here.

While `qtbases` is not yet as mature as `tcltk` and `RGtk2`, we include it in this book, as Qt compares favorably to GTK+ in terms of GUI features and excels in several areas, including its fast graphics canvas and integration of the WebKit web browser.¹ In addition, Qt, as a commercially supported package, has thorough documentation of its API, including many C++ examples. However, the complexity of C++ and Qt may present some challenges to the R user. In particular, the developer should have a strong grounding in object-oriented programming and have a basic understanding of memory management.

The `qtbases` package is developed as part of the *qtinterfaces* project at R Forge: <https://r-forge.r-project.org/projects/qtinterfaces/>. It depends on the Qt framework, available as a binary install from <http://qt.nokia.com/>.

10.3 Classes and objects

The `qtbases` package exports very few objects. The central one is an environment, `Qt`, that represents the Qt library in R.² The components of this environment are `RQtClass` objects that represent an actual C++ class or namespace. For example, the `QWidget` class is represented by `Qt$QWidget`:

```
Qt$QWidget
```

Class 'QWidget' with 320 public methods

An `RQtClass` object contains methods in the class scope (*static* methods in C++), enumerations defined by the class, and additional `RQtClass` objects representing nested classes or namespaces. Here we list some of the components of `QWidget` and access one of the enumeration values:

```
head(names(Qt$QWidget), n = 3)
```

¹There is a GTK+ WebKit port, but it is not included with GTK+ itself.

² The `Qt` object is an instance of `RQtLibrary`. The `qtbases` package provides infrastructure for binding any conventional C++ library, even those independent of Qt. Third party packages can define their own `RQtLibrary` object for some other library.

10. PROGRAMMING GUIs USING QT

```
[1] "connect"          "DrawChildren"      "DrawWindowBackground"
```

```
Qt$QWidget$DrawChildren
```

```
Enum value: DrawChildren (2)
```

Most importantly, however, an instance of `RQtClass` is in fact an R function object, and serves as the constructor of instances of the class. For example, we could construct an instance of `QWidget`:

```
w <- Qt$QWidget()
```

The `w` object has a class structure that reflects the class inheritance structure of Qt:

```
class(w)
```

```
[1] "QWidget"          "QObject"          "QPaintDevice"
[4] "UserDefinedDatabase" "environment"      "RQtObject"
```

The base class, `RQtObject`, is an environment containing the properties and methods of the instance. For `w`, we list the first few using `ls`:

```
head(ls(w))
```

```
[1] "setShown"          "releaseShortcut"
[3] "actions"           "setAccessibleDescription"
[5] "isWindowModified"  "size"
```

Properties and methods are accessed from the environment in the usual manner. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window:

```
w$windowTitle          # initially NULL
```

```
NULL
```

```
w$windowTitle <- "a new title"
w$windowTitle
```

```
[1] "a new title"
```

Although Qt defines methods for accessing properties, the R user will normally invoke methods that perform some action. For example, we could show our widget:

```
w$show()
```

The environment structure of the object masks the fact that the properties and methods may be defined in a parent class of the object. For example, a button widget is provided by the `QPushButton` constructor, as in

```
b <- Qt$QPushButton()
```

QPushButton extends QWidget and thus inherits the properties like windowTitle:

```
is(b, "QWidget") # Yes
```

```
[1] TRUE
```

```
b$windowTitle
```

```
NULL
```

It is important to realize this distinction when referencing the documentation. As with GTK+, the methods are documented with the class that declares the method.

10.4 Methods and dispatch

In C++, it is possible to have multiple methods and constructors with the same name, but different signatures. This is called *overloading*. An overloaded method is roughly similar to an S4 generic, save the obvious difference that an S4 generic does not belong to any class. The selected overload is that with the signature that best matches the types of the arguments. The exact rules of overload resolution are beyond our scope.

It is particularly common to overload constructors. For example, a simple push button can be constructed in several different ways. Here again is the invocation of the QPushButton constructor with no arguments:

```
b <- Qt$QPushButton()
```

By convention, all classes derived from QObject, including QWidget, provide a constructor that accepts a parent QObject. This has important consequences that are discussed later. We demonstrate this for QPushButton:

```
w <- Qt$QWidget()
b <- Qt$QPushButton(w)
```

An alternative constructor for QPushButton accepts the text for the label on the button:

```
b <- Qt$QPushButton("Button text")
```

Buttons may also have icons, for example

```
icon <- Qt$QIcon(system.file("images/ok.gif", package="gWidgets"))
b <- Qt$QPushButton(icon, "Ok")
```

We have passed three different types of object as the first argument to Qt\$QPushButton: a QWidget, a string, and finally a QIcon. The dispatch depends only on the

type of argument, unlike the constructors in RGtk2, which dispatch based on which arguments are specified. (In particular, dispatch is based on position of argument, but not on names given to arguments. We use names only for clarity in our examples.)

10.5 Properties

Every `QObject`, which includes every widget, may declare a set of properties that represent its state. We list some of the available properties for our button:

```
head(qproperties(b))
```

	type	readable	writable
objectName	QString	TRUE	TRUE
modal	bool	TRUE	FALSE
windowModality	Qt::WindowModality	TRUE	TRUE
enabled	bool	TRUE	TRUE
geometry	QRect	TRUE	TRUE
frameGeometry	QRect	TRUE	FALSE

As shown in the table, every property has a type and logical settings for whether the property is readable and/or writeable. Virtually every property value may be read, and it is common for properties to be read-only. For example, we can fully manipulate the `objectName` property, but our attempt to modify the `modal` property fails:

```
b$objectName <- "My button"
b$objectName
```

```
[1] "My button"
```

```
b$modal
```

```
[1] FALSE
```

```
try(b$modal <- TRUE)
```

Qt provides accessor methods for getting and setting properties. The getter methods have the same name as the property, so they are masked at the R level. Setter methods are available and are typically named with the word "set" followed by the property name:

```
b$setObjectName("My button")
```

However, it is recommended to use the replacement syntax shown in the previous example, for the sake of symmetry.

10.6 Signals

Qt in C++ uses an architecture of signals and slots to have components communicate with each other. A component emits a signal when some event happens, such as a user clicking on a button. Qt allows one to define a special type of method known as a slot in another component (or the same) that can be connected to the signal as the handler. The two components are decoupled as the emitter does not need to know about the receiver except through the signal connection. In R, any function can be treated as a slot and connected as a signal handler. This is similar to the signal handling in RGtk2. The function `qconnect` establishes the connection of an R function to a signal. For example

```
b <- Qt$QPushButton("click me")
qconnect(b, "clicked", function() print("ouch"))
b$show()
```

Signals are defined by a class and are inherited by subclasses. Here, we list some of the available signals for the `QPushButton` class:

```
tail(qsignals(Qt$QPushButton))
```

	name	signature
3	<code>customContextMenuRequested</code>	<code>customContextMenuRequested(QPoint)</code>
4	<code>pressed</code>	<code>pressed()</code>
5	<code>released</code>	<code>released()</code>
6	<code>clicked</code>	<code>clicked(bool)</code>
7	<code>clicked</code>	<code>clicked()</code>
8	<code>toggled</code>	<code>toggled(bool)</code>

The signal definition specifies the callback signature, given in the `signature` column. Like other methods, signals can be overloaded so that there are multiple signatures for a given signal name. Signals can also have default arguments, and arguments with a default value are optional in the signal handler. We see this for the `clicked` signal, where the `bool` (logical) argument, indicating whether the button is checked, has a default value of `FALSE`. The `clicked` signal is automatically overloaded with a signature without any arguments.

The `qconnect` function attempts to pick the correct signature by considering the number of formal arguments in the callback. When two signatures have the same number of arguments, one will be chosen arbitrarily. To connect to a specific signature, the full signature, rather than only the name, should be passed to `qconnect`. For example, the following is equivalent to the invocation of `qconnect` above:

```
qconnect(b, "clicked(bool)", function(checked) print("ouch"))
```

Any object passed to the optional argument `user.data` is passed as the last argument to the signal handler. The user data serves to parameterize

the callback. In particular, it can be used to pass in a reference to the sender object itself.

The `qconnect` function returns a dummy `QObject` instance that provides the slot that wraps the R function. This dummy object can be used with the `disconnect` method on the sender to break the signal connection:

```
proxy <- qconnect(b, "clicked", function() print("ouch"))
b$disconnect(proxy)
```

```
[1] TRUE
```

One can block all signals from being emitted with the `blockSignals` method, which takes a logical value to toggle whether the signals should be blocked.

Unlike GTK+, Qt widgets generally do not emit hardware events, such as a mouse press, via signals. Instead, a method in the widget is invoked upon receipt of an event. The developer is expected to extend the widget and override the method to catch the event. The apparent philosophy of Qt is that hardware events are low-level and thus should be handled by the widget, not some other instance. We will discuss extending classes in Section ??.

Example 10.2 demonstrates handling widget events.

10.7 Enumerations and flags

Often, it is useful to have discrete variables with more than two states, in which case a logical value is no longer sufficient. For example, the label widget has a property for how its text is aligned. It supports the alignment styles left, right, center, top, bottom, etc. These styles are enumerated by integer values and Qt defines these by name within the relevant class or, for global enumerations, in the Qt namespace. Here are examples of both:

```
Qt$Qt$AlignRight
```

```
Enum value: AlignRight (2)
```

```
Qt$QSizePolicy$Expanding
```

```
NULL
```

The first is the value for right alignment from the `Alignment` enumeration in the Qt namespace, while the second is from the `Policy` enumeration in the `QSizePolicy` class (referenced here by `QSizePolicy::Policy`).

Although these enumerations can be specified directly as integers, they are given the class `QtEnum` and have the overloaded operators `|` and `&` to combine values bitwise. This makes the most sense when the values correspond to bit flags, as is the case for the alignment style. For example, aligning the text in a label in the upper right can be done through

```
l <- Qt$QLabel("Our text")
l$alignment <- Qt$Qt$AlignRight | Qt$Qt$AlignTop
```

To check if the alignment is to the right, we could query by: ³

```
as.logical(l$alignment & Qt$Qt$AlignRight)
```

```
[1] FALSE
```

10.8 Defining Classes and Methods

As Qt is implemented in an object-oriented language, C++, the designers of the API expect the developer to extend Qt classes, like `QWidget`, during the normal course of GUI development. This is a significant difference from GTK+, where it is only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbase` package allows the R user to extend C++ classes in order to enhance the features of Qt.

We will demonstrate this functionality by example. In the introductory example of Section 10.1 we saw this minimal use of `qsetClass`. Our aim is to extend the `QValidator` class to restrict the input in a text entry (`QTextEdit`) to a valid date. The first step is to declare the class, and then methods are added individually to the class definition, in an analogous manner to the `methods` package. We start by declaring the class:

```
qsetClass("DateValidator", Qt$QValidator)
```

The class name is given as `DateValidator` and it extends the `QValidator` class in Qt. Note that only single inheritance is supported.

As a side-effect of the call to `qsetClass`, a variable named `DateValidator` has been assigned into the global environment (the scoping is similar to `methods::setClass`):

```
DateValidator
```

```
Class 'R::GlobalEnv::DateValidator' with 57 public methods
```

It is an error to redefine the class as above without first deleting the object.

To define a method on our class, we call the `qsetMethod` function:

```
qsetMethod("validate", DateValidator, validateDate)
```

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

The virtual method `validate` declared by `QValidator` has been overridden by the `DateValidator` class. The `validateDate` function implements the override and has been defined invisibly for brevity.

³As flags, combinations of enumerations, are not stored as integers they are not of class `QtEnum`, so the algebra of these operators is limited.

10. PROGRAMMING GUIs USING QT

As `DateValidator` is an `RQtClass` object, we can create an instance by invoking `DateValidator` like any other function:

```
validator <- DateValidator()
```

Now that we have our validator, we can use it with a text entry:

```
e <- Qt$QLineEdit()
v <- DateValidator(e)
e$setValidator(v)
```

```
NULL
```

```
e$show()
```

```
NULL
```

As Qt is implemented in an object-oriented language, C++, the designers of the API expect the developer to extend Qt classes, like `QWidget`, during the normal course of GUI development. This is a significant difference from GTK+, where it is only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbases` package allows the R user to extend C++ classes in order to enhance the features of Qt. The `qtbases` package includes functions `qsetClass` and `qsetMethod` to create subclasses and methods of subclasses.

To create a subclass of some class, say `QWidget`, the basic use is simply

```
qsetClass("SubClass", Qt$QWidget)
```

This assigns to the variable `SubClass` an object inheriting from `RQtClass`. (The `where` argument can be used to specify where this variable is defined, in a manner similar to S4's `setClass`.) Note the lack of parentheses for the parent object (`Qt$QWidget`), as the parent also inherits from `RQtClass`. These objects are constructors (functions) that have augmented environments containing the available methods and properties.

By default, the parent's constructor is inherited by the subclass. To create a new constructor, say to initialize values, or to make a compound widget, the constructor argument is used. This takes a function that will be called by positional arguments, so default values are used, but not by name. The function should allow a parent argument, to pass along a parent widget. A typical minimal usage might be

```
qsetClass("SubClass2", Qt$QWidget, function(parent=NULL) {
  super(parent)
  ## ... more here
})
```

Within the body of a constructor, the `super` variable refers to the parent's constructor, and in the above is used to set the parent object. Within the body of

a constructor the variable `this` refers to the environment of the instance of the subclass being constructed. To set a property, we can assign to a component. This environment augments the constructor's environment, so one can refer to the methods without a prefix. For example, a simple constructor to pass in a window title and some special property might look like

```
qsetClass("SubClass3", Qt$QWidget, function(title, prop, parent=NULL) {
  super(parent)
  this$property <- prop
  setWindowTitle(title)
})
```

One may define new methods, or override methods from a base class through the `qsetMethod` function. The arguments are the method name, the class the method is defined in, the method's definition, and an optional value for access indicating if the method should be "public", "protected", or "private".

Within the body of the method definition the variable `super` is a function, somewhat like `do.call`, that can be used to call a method from the parent class by name (the first argument) passing in all subsequent arguments. The functionality is similar to that provided by S4's `callNextMethod`. The variable `this` refers again to the instance's environment. Again for convenience this is placed into the search path of the function call.

For example, methods to create accessors for a property may be defined as

```
qsetClass("SubClass4", Qt$QWidget)
qsetMethod("property", SubClass4, function() property)
qsetMethod("setProperty", SubClass4, function(value) {
  this$property <- value
})
```

To override a base method by adding to its call can be done with the `super` function. For example, to store the window title in our property one might do this:

```
qsetMethod("setWindowTitle", SubClass4, function(value) {
  setProperty(value)
  super("setWindowTitle", value)
})
```

Example 10.1: A "error label"

A common practice when validation is used for text entry is to have a "buddy label." That is an accompanying label to set an error message. As Qt uses "buddy" for something else, we call this an "error label" below. We show how to implement such a widget in `qtbase` where we define a new type of widget that combines a `QTextEdit` and the `QLabel` for reporting errors.

This demonstrates the definition of a custom constructor for an R class. The R function implementing the constructor must be passed during the call to `qsetClass`:

```
qsetClass("ErrorLabel", Qt$QLineEdit,
  function(text, parent=NULL, message="") {
    super(parent)

    this$widget <- Qt$QWidget()           # for attaching
    this$error <- Qt$QLabel()             # set height=0
    this$error$setStyleSheet("* {color: red}") # set color

    lyt <- Qt$QGridLayout()               # layout
    widget$setLayout(lyt)

    lyt$setVerticalSpacing(0)
    lyt$addWidget(this, 0, 0, 1, 1)
    lyt$addWidget(error, 1, 0, 1, 1)

    if(nchar(message) > 0)
      setMessage(message)
    else
      setErrorHeight(FALSE)
    if(!missing(text)) setText(text)
  })
```

By convention, every widget in Qt accepts its parent as an argument to its constructor. Via `super`, the parent is passed to the base class constructor and on up the hierarchy. The `super` function does not exist outside the scope of a constructor (or method). Within a constructor, `super` invokes the constructor of the parent (super) class, with the given arguments. Within a method implementation, `super` will call a method (named in the first parameter) in the parent class (see Example 10.2).

In addition to the call to `super`, we define a `QWidget` instance to contain the line edit widget and its label. These are placed within a grid layout. The use of `this` refers to the object we are creating. The new method `setErrorHeight` is used to flatten the height of the label when it is not needed and is defined below. The final line sets the initial text in the line edit widget. The R environment where `setText` is defined is extended by the environment of this constructor, so no prefix is needed in the call.

The widget component is needed to actually show the widget. We create an accessor method:

```
qsetMethod("widget", ErrorLabel, function() widget)
```

We extend the API of the line edit widget for this subclass to modify the message. We define three methods, one to get the message, one to set it and a convenience function to clear the message.

```
qsetMethod("message", ErrorLabel, function() error$text)
qsetMethod("setMessage", ErrorLabel, function(msg="") {
  if(nchar(msg) > 0)
    error$setText(msg)
  setErrorHeight(nchar(msg) > 0)
})
qsetMethod("clear", ErrorLabel, function() setMessage())
```

Finally, we define the method to set the height of the label, so that when there is no message it has no height. We use a combination of setting both the minimum and maximum height.

```
qsetMethod("setErrorHeight", ErrorLabel, function(do.height=FALSE)
{
  if(do.height) {
    m <- 18; M <- 100
  } else {
    m <- 0; M <- 0
  }
  error$setMinimumHeight(m)
  error$setMaximumHeight(m)
})
```

To use this widget, we have the extra call to `widget()` to retrieve the widget to add to a GUI. In the following, we just show the widget.

```
e <- ErrorLabel()
w <- e$widget()           # get widget to show
w$show()                  # to view widget
e$setMessage("A label")   # opens message
e$clear()                 # clear message, shrink space
```

10.9 Common methods for QWidgets and QObjects

The widgets we discuss in the sequel inherit many properties and methods from the base `QObject` and `QWidget` classes. The `QObject` class is the base class and forms the basis for the object heirarchy and the event processing system. The `QWidget` class is the base class for objects with a user interface. Defined in this class are several methods inherited by the widgets we discuss.

The function `qmethods` will show the methods defined for a class. It returns a data frame with variables indicating the name, return value, signature, and whether the method is protected and static. For example, for a simple button we have many methods, of which we can only discuss a handful.

```
out <- qmethods(Qt$QPushButton)
dim(out)                               # many methods
```

Showing or hiding a widget Widgets must have their `show` method called in order to have them drawn to the screen. This call happens through the `print` method for an object inheriting from `QWidget`, but more typically is called recursively by Qt when showing the children as a top-level window is drawn. The method `raise` will raise the window to the top of the stack of windows, in case it is covered. The method `hide` will hide the widget.

A widget can also be hidden by calling its `setVisible` method with a value of `FALSE` and reshowed using a value of `TRUE`. Similarly, the method `setEnabled` can be used to toggle whether a widget is sensitive to user input, including mouse events.

Only one widget can have the keyboard focus. This is changed by the user through tab-navigation or mouse clicks (unless customized, see `focusPolicy`), but can be set programatically through the `setFocus` method, and tested through the `hasFocus` method.

Qt has a number of means to notify the user about a widget when the mouse hovers over it. The `setToolTip` method is used to specify a tooltip as a string. The message can be made to appear in the status bar of a top-level window through the method `setStatusTip`.

The size of a widget A widget may be drawn with its own window, or typically embedded in a more complicated GUI. The size of the widget can be adjusted through various methods.

First, we can get the size of the widget through the methods `frameGeometry` and `frameSize`. The `frameGeometry` method returns a `QRect` instance, Qt's rectangle class for integer sizes. Rectangles are parameterized by an $x - y$ position and two dimensions (x , y , `width` and `height`). In this case, the position refers to the upper left coordinate and dimensions are in pixels. The convenience function `qrect` is provided to construct `QRect` instances. The `frameSize` method returns a `QSize` object with properties `width` and `height`. The `qsize` function is a convenience constructor for objects of this class.

The widget's size can be adjusted several ways: with the `resize` method, by modifying the rectangle and then resetting the geometry with `setGeometry`, or directly through the same method when integer values are given for the arguments.

```
w <- Qt$QWidget()
rect <- w$frameGeometry
rect$width()
rect$setWidth(2 * rect$width())
w$setGeometry(rect)
```

Although the above sets the size, it does not fix it. If that is desired, the methods `setFixedSize` or `setFixedWidth` are available.

When a widget is resized, one can constrain how it changes by specifying a minimum size or maximum size. These values work in combination with the size policy of the widget. The properties `minimumSize`, `minimumWidth`, `minimumHeight`, `maximumSize`, `maximumWidth` and `maximumHeight`, and their corresponding setters, are the germane ones. How these get used is determined by the `sizePolicy` property. For example, buttons will only grow in the x direction – not the y direction due to their default size policy.

The method `update` is used to request a repainting of a widget and the method `updateGeometry` is used to notify any layout manager that the size hint has changed.

Fonts

Fonts in Qt are handled through the `QFont` class. In addition to the basic constructor, one constructor allows the programmer to specify a family, such as `helvetica`; pointsize, an integer; weight, an enumerated value such as `Qt::QFont::Light` (or `Normal`, `DemiBold`, `Bold`, or `Black`); and whether the italic version should be used, as a logical.

For example, a typical font specification may be given as follows:

```
f <- Qt::QFont(family="helvetica", ps=12,
               weight=Qt::QFont::Bold,
               italics=TRUE)
```

For widgets, the `setFont` method can be used to adjust the font. For example, to change the font for a label we have

```
l <- Qt::QLabel("Text for the label")
l$setFont(f)
```

The `QFont` class has several methods to query the font and to adjust properties. For example, there are the methods `setFamily`, `setUnderline`, `setStrikeout` and `setBold` among others.

Styles

Qt uses styles to provide a means to customize the look and feel of an application for the underlying operating system. It is generally recommended that a GUI be consistent with the given style. For example, each style implements a palette of colors to indicate the states “active” (has focus), “inactive” (does not have focus), and “disabled” (not sensitive to user input). Although, many widgets do not have a visible distinction between active or inactive. The role an object plays determines the type of coloring it should receive. A palette has an enumeration in `QPalette::ColorRole`. Sample ones are `Qt::QPalette::Highlight`, to indicate a selected item, or `Qt::QPalette::WindowText` to indicate a foreground color.

These roles are used for setting the foreground or background role to give a widget a different look, as illustrated in Example 10.2.

Style Sheets

Cascading style sheets (CSS) are used by web designers to decouple the layout and look and feel of a web page from the content of the page. Qt implements the mechanism in the `QWidget` class to customize a widget through the CSS syntax. The implemented syntax is described in the overview on stylesheets provided with Qt documentation and is not summarized here, as it is quite readable.

To implement a change through a style sheet involves the `setStyleSheet` method. For example, to change the background and text color for a button we could have

```
b <- Qt$QPushButton("Style sheet example")
b$show()
b$setStyleSheet("QPushButton {color: red; background: white;}")
```

One can also set a background image:

```
ssheet <- sprintf("* {background-image: url(%s)}", "logo.png")
b$setStyleSheet(ssheet)
```

10.10 Drag and drop

Qt has native support for basic drag and drop activities for some of its widgets, such as text editing widgets, but for more complicated situations such support must be programmed in. The toolkit provides a clear interface to allow this.

A drag and drop event consists of several stages: the user selects the object that initiates the drag event, the user then drags the object to a target, and finally a drop event occurs. In addition, several decisions must be made, e.g., will the object “move” or simply be copied. Or, what kind of object will be transferred (an image? text?, ...) etc. The type is specified using a standard MIME specification.

Initiating a drag and drop source When a drag and drop sequence is initiated, the widget receiving the mouse press event needs to set up a `QDrag` instance that will transfer the necessary information to the receiving widget. In addition, the programmer specifies the type of data to be passed, as an instance of the `QMimeData` class. Finally, the user must call the `exec` method with instructions indicating what happens on the drop event (the supported actions) and possibly what happens if no modifier keys are specified. These are given using the enumerations `CopyAction`, `MoveAction`, or `LinkAction`.

This is usually specified in a method for the `mousePressEvent` event, so is done in a subclass of the widget you wish to use.

Creating a drop target The application must also set up drop sources. Each source has its method `setAcceptDrops` called with a `TRUE` value. In addition, one must implement several methods so again, a subclass of the desired widget is needed. Typically one implements at a minimum a `dropEvent` method. This method has an `QDropEvent` instance passed in which has the method `mimeData` to get the data from the `QDrag` instance. This data has several methods for querying the type of data, as illustrated in the example. If everything is fine, one calls the event's `acceptProposedAction` method to set the drop action. One can also specify other drop actions.

Additionally, one can implement methods for `dragMoveEvent` and `dragLeaveEvent`. In the example, the move and leave event adjust properties of the widget to indicate it is a drop target.

Example 10.2: Drag and drop

We will use subclasses of the label class to illustrate how one implements basic drag-and-drop functionality. Our treatment follows the Qt tutorial on the subject. We begin by setting up a label to be a drag target.

```
qsetClass("DragLabel", Qt$QLabel, function(text="", parent=NULL) {
  super(parent)
  setText(text)

  setAlignment(Qt$Qt$AlignCenter)
  setMinimumSize(200, 200)
})
```

The main method to implement for the subclass is `mousePressEvent`. Its argument `e` contains event information for the mouse press event, we don't need it here. We have the minimal structure here: implement mime data to pass through, set up a `QDrag` instance for the data, then call the `exec` method to initiate the drag event. The `exec` method call has optional arguments to specify what action should be done. The default here is move, copy, link, for which only copy makes sense.

```
qsetMethod("mousePressEvent", DragLabel, function(e) {
  md <- Qt$QMimeData()
  md$setText(text)

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)
```

```
drag$exec()  
})
```

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

Implementing a label as a drop target is a bit more work, as we customize its appearance. Our basic constructor follows:

```
qsetClass("DropLabel", Qt$QLabel, function(text="", parent=NULL) {  
  super(parent)  
  
  setText(text)  
  setAcceptDrops(TRUE)  
  
  this$bgrole <- backgroundRole()  
  setAlignment(Qt$Qt$AlignCenter)  
  setMinimumSize(200, 200)  
  setAutoFillBackground(TRUE)  
  clear()  
})
```

We wish to override the call to `setText` above, as we want to store the original text in a property of the widget. Note the use of `super` with a method definition below to call the next method.

```
qsetMethod("setText", DropLabel, function(str) {  
  this$orig_text <- str  
  super("setText", str) # next method  
})
```

The `clear` method is used to restore the label to an initial state. We have saved the background role and original text as properties for this purpose.

```
qsetMethod("clear", DropLabel, function() {  
  setText(this$orig_text)  
  setBackgroundRole(this$bgrole)  
})
```

The enter event notifies the user that a drop can occur on this target by changing the text and the background role.

```
qsetMethod("dragEnterEvent", DropLabel, function(e) {  
  super("setText", "<Drop Text Here>")  
  setBackgroundRole(Qt$QPalette$Highlight)  
  
  e$acceptProposedAction()  
})
```

The move and leave events are straightforward. We call `clear` when the drag leaves the target to restore the widget.


```
qsetMethod("dragMoveEvent", DropLabel, function(e) {
  e$acceptProposedAction()
})
qsetMethod("dragLeaveEvent", DropLabel, function(e) {
  clear()
#  e$acceptProposedAction()
})
```

Finally, the important drop event. The following shows how to implement this in more generality than is needed for this example, as we only have text in our mime data.

```
qsetMethod("dropEvent", DropLabel, function(e) {
  md <- e$mimeTypeData()

  if(md$hasImage()) {
    setPixmap(md$imageData())
  } else if(md$hasHtml()) {
    setText(md$html)
    setTextFormat(Qt$Qt$RichText)
  } else if(md$hasText()) {
    setText(md$text())
    setTextFormat(Qt$Qt$PlainText)
  } else {
    setText("No match") # replace ...
  }

  setBackgroundRole(this$bgrole)
  e$acceptProposedAction()
})
```


Layout managers

Qt provides a set of classes to facilitate the layout of child widgets of a component. These layout managers, derived from the `QLayout` class, are tasked with determining the geometry of child widgets, according to a specific layout algorithm. Layout managers will generally update the layout whenever a parameter is modified, a child widget is added or removed, or the size of the parent changes. Unlike GTK+, where this management is tied to a container object, Qt decouples the layout from the widget.

In this chapter we discuss the basic layouts and end with a discussion on the `QMainWindow` class for top-level windows.

We begin with an example that shows many of the different types of layouts.

Example 11.1: Using layout managers to mock up an interface

This example illustrates how to layout a somewhat complicated GUI by hand using a combination of different layout managers provided by Qt. Figure 11.1 shows a screenshot of the finished layout.

Qt provides standard layouts for box layouts and grid layouts, in addition there are notebook containers and special layouts for forms, all seen in the following.

Our GUI is layed out from the outside in. The first layout used is a grid layout which will hold three main areas: one for a table (we use a label for now), one for a notebook, and a layout to hold some buttons.

A `QWidget` instance can hold one immediate layout set by the `setLayout` method. We use a widget for a top-level window and begin by specifying a grid layout.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
gridLayout <- Qt$QGridLayout()
w$setLayout(gridLayout)
```

Here we define the two main widgets and the layout for our buttons.

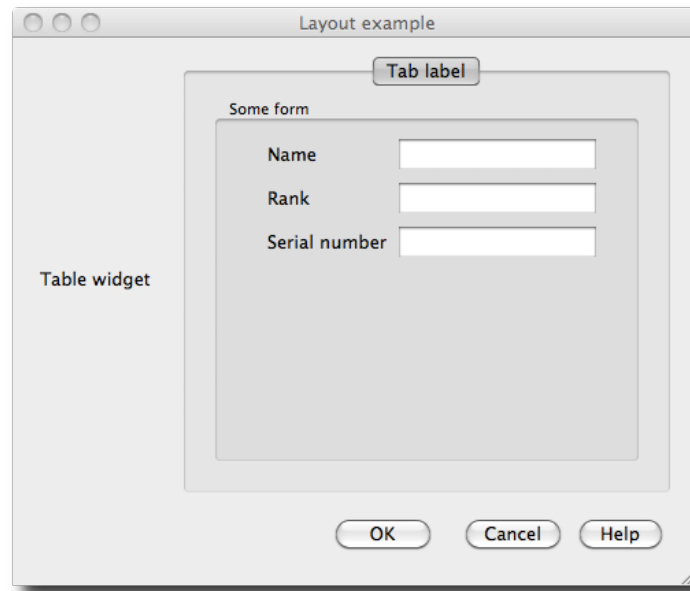


Figure 11.1: A mock GUI illustrating various layout managers provided by Qt.

```
tableWidget <- Qt$QLabel("Table widget")
nbWidget <- Qt$QTabWidget()
buttonLayout <- Qt$QHBoxLayout()
```

Grid layouts have two main methods, `addWidget` which is inherited from the base `QLayout` class, and `addLayout`. To use them, we specify what part of the grid the child widget or layout will occupy through a 0-based row and column and optionally a specification of the row and column span.

```
gridLayout$addWidget(tableWidget, row=0, column=0,
                      rowspan=1, colspan=1)
gridLayout$addWidget(nbWidget, 0, 1)
gridLayout$addLayout(buttonLayout, 1, 1)
```

When resizing, we want to give the area to the notebook widget in row 1, column 1. We do this by adding a weight to the stretch value.

```
gridLayout$setRowStretch(1, stretch=5)
```

```
NULL
```

```
gridLayout$setColumnStretch(1, stretch=5)
```

```
NULL
```

Buttons are added to our box layout through the `addWidget` method. In this case, we want to push the buttons to the right side of the GUI, so we first add a stretch. Stretches are specified by integers. Unallocated space is given first to widgets that have a non-zero stretch factor. We also set spacing of 12 pixels between the “OK” and “Cancel” buttons.

```
b <- sapply(c("OK", "Cancel", "Help"),
            function(i) Qt$QPushButton(i))
buttonLayout$addStretch(1L)
buttonLayout$addWidget(b$OK)
buttonLayout$addSpacing(12L)
buttonLayout$addWidget(b$Cancel)
buttonLayout$addWidget(b$Help)
```

For our notebook widget we add pages through the `addTab` method. We pass in the child widget and a label below. In addition, we set a tooltip for the tab label to give more feedback to the user.

```
nbPage <- Qt$QWidget()
nbWidget$addTab(nbPage, "Tab label")
nbWidget$setTabToolTip(0, "A notebook page with a form")
```

We wish to layout a form inside of the notebook tab, but first create a framed widget using a `QGroupBox` widget to hold the form layout. This widget allows us to easily specify a title. We add this to the page using a box layout.

```
f <- Qt$QGroupBox()
f$setTitle("Some form")
#
lyt <- Qt$QHBoxLayout()
nbPage$setLayout(lyt)
lyt$addWidget(f)
```

The form layout allows us to create standardized forms where each row contains a label and a widget. Although, this could be done with a grid layout, using the form layout is more convenient, and allows Qt to style the page as appropriate for the underlying operating system.

```
formLayout <- Qt$QFormLayout()
f$setLayout(formLayout)
```

Our form template just uses 3 line-edit widgets. The `addRow` method makes it easy to specify the label and the widget.

```
l <- sapply(c("name", "rank", "snumber"), function(i) Qt$QLineEdit())
formLayout$addRow("Name", l$name)
formLayout$addRow("Rank", l$rank)
formLayout$addRow("Serial number", l$snumber)
```

Finally, we set the minimum size for our GUI and call `show` on the top-level widget.

```
w$setMinimumSize(width=500, height=400)
w$show()
```

11.1 Box layouts

Box layouts arrange child widgets by packing in values horizontally or vertically. The `QHBoxLayout` class implements a horizontal layout whereas `QVBoxLayout` provides a vertical one. Both of these classes subclass the `QBoxLayout` class where most of the functionality is documented. The `direction` property specifies how the layout is done. By default, this is left to right or top to bottom, but can be set (e.g., using `Qt::LeftToRight`).

Child widgets are added to a box container through the `addWidget` method. The basic call specifies just the child widget, but one can specify an integer value for `stretch` and an alignment enumeration. In addition to adding child widgets, one can nest child layouts through `addLayout`.

A count of child widgets is returned by `count`. Some methods use an index (0-based) to refer to the child components. For example, The `insertWidget` can be used to insert a widget, with arguments similar to `addWidget`. Its initial argument is an integer specifying the index. All child widgets with this index or higher have their index increased by 1. Internally, layouts store child components as items of class `QLayoutItem`. The item at a given numeric index is returned by `itemAt`. The actual child component is retrieved by passing the item to the layout's `widget` method.

Qt provides the methods `removeItem` and `removeWidget` to remove a widget from a layout. Once removed from one layout, these may be reparented if desired, or destroyed. This is done by setting the widget's parent to `NULL` using `setParent`.

Size and space considerations The allocation of space to child widgets depends on several factors.

The Qt documentation for layouts spells out the steps: ¹

1. All the widgets will initially be allocated an amount of space in accordance with their `sizePolicy` and `sizeHint`.
2. If any of the widgets have stretch factors set, with a value greater than zero, then they are allocated space in proportion to their stretch factor.
3. If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an Expanding size policy first.
4. Any widgets that are allocated less space than their minimum size (or minimum size hint if no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum

¹<http://doc.qt.nokia.com/4.6/layout.html>

Table 11.1: Possible size policies

Policy	Meaning
Fixed	to require the size hint exactly
Minimum	to treat the size hint as the minimum, allowing expansion
Maximum	to treat the size hint as the maximum, allowing shrinkage
Preferred	to request the size hint, but allow for either expansion or shrinkage
Expanding	to treat like Preferred, except the widget desires as much space as possible
MinimumExpanding	to treat like Minimum, except the widget desires as much space as possible
Ignored	to ignore the size hint and request as much space as possible

size or minimum size hint in which case the stretch factor is their determining factor.)

- Any widgets that are allocated more space than their maximum size are allocated the maximum size space they require. (Widgets do not have to have a maximum size in which case the stretch factor is their determining factor.)

Every widget returns a size hint to the layout from the `sizeHint` method/property. The interpretation of the size hint depends on the `sizePolicy` property. The size policy is an object of class `QSizePolicy`. It contains a separate policy value, taken from the `QSizePolicy` enumeration, for the vertical and horizontal directions. The possible size policies are listed in Table ??.

tab:qt:size-policies As an example, consider `QPushButton`. It is the convention that a button will only allow horizontal, but not vertical, expansion. It also requires that it has enough space to display its entire label. Thus a `QPushButton` instance returns a size hint depending on the label dimensions and has the policies `Fixed` and `Minimum` as its vertical and horizontal policies respectively. We could prevent a button from expanding at all:

```
b <- Qt::QPushButton("No expansion")
b$setSizePolicy(vertical=Qt::QSizePolicy$Fixed,
                horizontal=Qt::QSizePolicy$Fixed)
```

Thus, the sizing behavior is largely inherent to the widget, rather than any layout parameters. This is a major difference from GTK+, where a widget can only request a minimum size and all else depends on the parent container widget. The Qt approach seems better at encouraging consistency in the layout behavior of widgets of a particular type.

Stretch factors are used to proportionally allocate space to widgets when they expand. When more than one widget can expand, how the space gets allocated is determined by their stretch factors (for box layouts set by the second argument of the `addWidget` method).

When a widget is allocated additional space, its alignment is determined by an alignment specification. This can be done when the widget is added, or later through the `setAlignment` method of the layout. The alignment is specified as a flag using the `Qt::AlignmentFlag` enumeration.

Spacing For a box layout, the space between two children is controlled through the `setSpacing` method. This sets the common width in pixels, which can be adjusted individually through the `addSpacing` method. The margin area around all the children can be adjusted with the `setContentsMargins` method, although this is often specified through the style.

Springs and Struts A stretchable blank widget can be added through the `addStretch` method, where an integer is specified to indicate the stretch factor. If no other widgets have a stretch specified then this widget will take all the non-requested space.

A strut (`addStrut`) can be specified in pixels to restrict the dimension of the box to a minimum height (or width for vertical boxes).

Scrolling layouts

It may be desirable to constrain the size of a layout and allow the user to pan through its children with scrollbars. The `QScrollArea` class makes this very straightforward, as you simply place a container widget into the scroll area. The method `addWidget` is used to add the child widget.

```
sa <- Qt$QScrollArea()
w <- Qt$QWidget()
sa$addWidget(w)
## no add to the child widget w
w$setMinimumSize(400,400)
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
for(i in rownames(state.x77)) {
  msg <- sprintf("%s had a population of %s thousand in 1977",
                 i, state.x77[i,"Population"])
  lyt$addWidget(Qt$QLabel(msg))
}
```

To ensure that a given child widget is visible, the method `ensureWidgetVisible` is available, where the widget is passed as the argument. So, to ensure the value for New York (row 32 in the data set) is visible, we have:


```
widget <- lyt$itemAt(32 - 1)$widget()
sa$ensureWidgetVisible(widget)
```

Framed Layouts

A frame with a title is a common decoration to a container often utilized to group together widgets that are naturally related. In Qt the `QGroupBox` class provides a container widget which can then hold a layout for child items. Its method `setTitle` can be used to set the title, or it can be passed to the constructor. If the standard position of the title determined from the style is not to the liking, it can be adjusted through the `setAlignment` method. This method takes from the alignment enumeration, for example `Qt::AlignLeft`, `Qt::AlignHCenter` or `Qt::AlignRight`. The property `flat` can be set to `TRUE` to minimize the allocated space.

Group boxes have a `checkable` property that if enabled the widget will be drawn with a checkbox to control whether the children of the group box are sensitive to user input.

Separators

The `QGroupBox` widget provides a framed area to separate off related parts of GUI. Sometimes, just a separating line is all that is desired. There is no separate separator widget in Qt, unlike GTK+. However, the `QFrame` class provides a general method for framed widgets (such as a label) that can be used for this purpose with the appropriate settings.

The frame shape can be a box or other types. For this purpose a line is desired. The enumerations `Qt::FrameHLine` or `Qt::FrameVLine` can be passed to the method `setFrameShape` to specify a horizontal or vertical line. Its appearance can be altered by `setFrameShadow`. A value of `Qt::FrameSunken` or `Qt::FrameRaised` is suitable.

11.2 Grid Layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns.

The `addWidget` method is used to add a child widget to the layout and the `addLayout` method creates nested layouts. Both methods have similar arguments. There are two methods. One where one specifies the child, and just a row and column index and optionally an alignment and another where the row and column indices are followed by specifications for the row and column span followed by an optional alignment.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column. The `item` method `widget` is

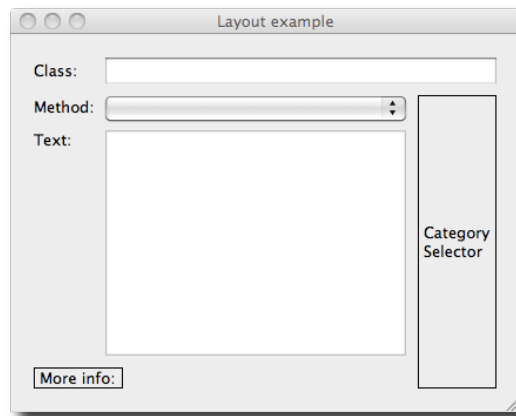


Figure 11.2: A mocked up layout using the `QGridLayout` class.

used to find the corresponding widget. Removing a widget is similar for a box layout using `removeItem` or `removeWidget`. The methods `rowCount` and `columnCount` can be used to find the current size of the grid.

Rows and columns are dual concepts and are so implemented. Consequently, both have similar methods differing only by the use of `column` or `row`. We discuss columns. A column minimum width can be set through `setColumnMinimumWidth`. If more space is available to a column than requested, then the extra space is apportioned according to the stretch factors. This can be set for a column through the `setColumnStretch` method. Taking an integer value 0 or larger.

The spacing between widgets can be set in both directions with `setSpacing`, or fine-tuned with `setHorizontalSpacing` or `setVerticalSpacing`. (The style may set these too wide for some tastes.)

Example 11.2: Using a grid layout

To illustrate grid layouts we mock up a GUI centered around a central text area widget (Figure 11.2). If the window is resized, we want that widget to get the extra space allocated to.

We begin by setting a grid layout for our parent widget.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
lyt <- Qt$QGridLayout()
w$setLayout(lyt)
```

We use the default left-alignment for labels in the following. Our first row has a label in column 1 and a text-entry widget spanning two columns.

```
lyt$addWidget(Qt$QLabel("Class:"), 0, 0)
lyt$addWidget(Qt$QLineEdit(), 0, 1, rowspan=1, colspan=2)
```

Our second row starts with a label and a combobox each spanning the default of one column.

```
lyt$addWidget(Qt$QLabel("Method:"), 1, 0)
lyt$addWidget(Qt$QComboBox(), 1, 1)
```

The third column of the second row and rest of the layout is managed by a sublayout, in this case a vertical box layout. We use a label as a stub and set a frame style to have it stand out.

```
lyt$addLayout(slyt <- Qt$QVBoxLayout(), 1, 2, rowspan=3, 1)
slyt$addWidget(l <- Qt$QLabel("Category\nSelector"))
l$setFrameStyle(Qt$QFrame$Box)
```

The text edit widget is added to the third row, second column. Since this widget will expand, we set an alignment for the label. Otherwise, the default alignment will center it in the vertical direction.

```
lyt$addWidget(Qt$QLabel("Text:"), 2, 0, Qt$Qt$AlignTop)
lyt$addWidget(l <- Qt$QTextEdit(), 2, 1)
```

Finally we add a space for information on the 4th row. Again we place this in a box. By default the box would expand to fill the space of the two columns, but we fix this below as an illustration.

```
lyt$addWidget(l <- Qt$QLabel("More info:"), 3, 0,
              rowspan=1, colspan=2)
l$setSizePolicy(Qt$QSizePolicy$Fixed, Qt$QSizePolicy$Preferred)
l$setFrameStyle(Qt$QFrame$Box)
```

As it is, since there are no stretch factors set, the space allocated to each row and column would be identical when resized. To force the space to go to the text widget, we set a positive stretch factor for the 3rd row and 2nd column. Since the others have the default stretch factor of 0, this will allow that cell to grow when the widget is resized.

```
lyt$setRowStretch(2, 1)
lyt$setColumnStretch(1, 1)
```

11.3 Form layouts

Forms can easily be generated with the grid layout, but Qt provides an even more convenient form layout (`QFormLayout`) that has the added benefit of conforming to the traditional styles for the underlying operating system. This can be used in combination with the `QDialogButtonBox` (Section 12.3), which provides a container for buttons that also tries to maintain an appearance consistent with the underlying operating system.

To add a child widget with a label is done through the `addRow` method, where the label, specified first, may be given as a string and the widget is specified second. The first argument can also be a widget to replace the label, and the second a layout for nesting layouts. The `insertRow` method is similar, only one first specifies the row number using a 0-based index. The `setSpacing` method can be used to adjust the default spacing between rows.

After construction, the widget may be retrieved through the `itemAt` method. This returns a layout item, to get the widget call its `widget` method. The `setWidget` method can be used to change a widget.

Example 11.3: Simple use of `QFormLayout`

The following illustrates a basic usage where three values are gathered.

```
w <- Qt$QWidget()
w$setWindowTitle("Example of a form layout")
w$setLayout(flyt <- Qt$QFormLayout())
l <- list()
flyt$addRow("mean", l$mean <- Qt$QLineEdit())
flyt$addRow("sd", l$sd <- Qt$QLineEdit())
flyt$addRow("n", l$n <- Qt$QLineEdit())
```

```
w$show(); w$raise()
```

```
NULL
```

```
NULL
```

The form style can be overridden using the `setFormAlignment` and `setLabelAlignment` methods. The Mac OS X default is to have center aligned form with right-aligned labels, whereas the Windows default is to have left-aligned labels. One can also override the default as to how the fields should grow when the widget is resized (`setFieldGrowthPolicy`).

For example,

```
flyt$setFormAlignment(Qt$Qt$AlignLeft | Qt$Qt$AlignTop)
flyt$setLabelAlignment(Qt$Qt$AlignLeft);
```

11.4 Notebooks

A notebook container is provided by the widget `QTabWidget`. This is not a layout, rather a notebook page consists of a label and widget. Of course, you can use a layout within the widget.

Pages are added through the method `addTab`. One can specify a widget; a widget and label; or a widget, icon and label. As well, pages may be inserted by index with the `insertTab` method.

Tabs allow the user to select from among the pages, and in Qt the tabs can be customized. The text for a tab is adjusted through `setTabText` and the icon through `setTabIcon`. These use a 0-based index to refer to the tab. A tooltip can be added through `setTabToolTip`. The tabs will have close buttons if the property `tabsClosable` is `TRUE`. One connects to the `tabCloseRequested` signal to actually close the tab. The tab position is adjusted through the `setTabPosition` method using values in the enumeration `QTabWidget::TabPosition`, which uses compass headings like `Qt::QTabWidget$North`. Calling `isMovable` with `TRUE` allows the pages to be reorganized by the user.

When there are numerous tabs, the method `setUsesScrollButtons` can indicate if the widget should expand to accommodate the labels or add scroll buttons.

The current tab is adjusted through the `currentIndex` property. The actual widget of the current tab is returned by `currentWidget`. To remove a page the `removeTab` is used, where tabs are referred to by index.

Example 11.4: A tab widget

A simple example follows. First the widget is defined with several properties set.

```
nb <- Qt::QTabWidget()
nb$setTabsClosable(TRUE)
nb$setMovable(TRUE)
nb$setUsesScrollButtons(TRUE)
```

We can add pages with a label or with a label and an icon:

```
nb$addTab(Qt::QPushButton("page 1"), "page 1")
icon <- Qt::QIcon("small-R-logo.jpg")
nb$addTab(Qt::QPushButton("page 2"), icon, "page 2")
## we add numerous tabs to see scroll buttons
for(i in 3:10) nb$addTab(Qt::QPushButton(i), sprintf("path %s", i))
```

The close buttons put out a request for the page to be closed, but do not handle directly. Something along the lines of the following is then also needed.

```
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
```

QStackedWidget

The `QStackedWidget` is provided by Qt to hold several widgets at once, with only one being visible. Similar to a notebook, only without the tab decorations to switch pages. For this widget, children are added through the `addWidget` method and can be removed with `removeWidget`. The latter

needs a widget reference. The currently displayed widget can be found from `currentWidget` (which returns `NULL` if there are no child widgets). Alternatively, one can refer to the widgets by index, with `count` returning their number, and `currentIndex` the current one and `indexOf` returns the index of the widget specified as an argument.

11.5 Paned Windows

Split windows with handles to allocate space are created by `QSplitter`. There is no limit on the number of child panes that can be created. The default orientation is horizontal, set the `orientation` property using `QtQtVertical` to change this.

Child widgets are added through the `addWidget` method. These widgets are referred to by index and can be retrieved through the `widget` method.

The `moveSplitter` method is not implemented, so programmatically moving the a splitter handle is not possible.

```
sp <- Qt$QSplitter()
sp$addWidget(Qt$QLabel("One"))
sp$addWidget(Qt$QLabel("Two"))
sp$addWidget(Qt$QLabel("Three"))
sp$setOrientation(Qt$Qt$Vertical)
```

```
sp$widget(0)$text          # text in first widget
```

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

11.6 Main windows

In Qt the `QMainWindow` class provides a widget for use as the primary widget in an interface. Although it is a subclass of `QWidget` – which we have used in our examples so far as a top-level window – the implementation also has preset areas for the standard menu bars, tool bars and status bars. As well, there is a possible dock area for “dockable widgets.”

A main window has just one central widget that is specified through the `setCentralWidget` method. The central widget can then have a layout and numerous children. The title of the window is set from the inherited method `setWindowTitle`.

Actions

The menubars and toolbars are representations of collections of actions, defined through the `QAction` class. As with other toolkits, an action encapsulates a command that may be shared among parts of a GUI, in this case

menu bars, toolbars and keyboard shortcuts. In Qt the action can hold the label (`setText`), an icon (`setIcon`), a status bar tip (`setStatusTip`), a tool tip (`setToolTip`), and a keyboard shortcut (`setShortcut`). The text and icon may be set at construction time, in addition to using the above methods.

Actions inherit the `enabled` method to toggle whether an action is sensitive to user input. Actions emit a `triggered` signal when activated (sending to the callback a boolean value for checked when applicable).

Keyboard shortcuts A keyboard shortcut use a `QKeySequence` to bind a key sequence to the command. Key sequences can be found from the standard shortcuts provided in the enumeration `Qt::QtKeySequence`, for example

```
Qt::QtKeySequence::Cut
```

```
Enum value: Cut (8)
```

This value (or more simply the "Cut" string) can be passed to the constructor to create the shortcut.

Using these standard shortcuts ensures that the keyboard shortcut is the standard one for the underlying operating system. Alternatively, custom shortcuts can be used, such as

```
Qt::QKeySequence("Ctrl-X, Ctrl-C")
```

```
QKeySequence instance
```

This shows how a modifier can be specified (from "Ctrl") a key (case insensitive), and how a comma can be used to create multi-key shortcuts.

For buttons and labels, a shortcut key can be specified by prefixing the text with an ampersand `&`, such as

```
button <- Qt::QPushButton("&Save")
```

Then the shortcut will be Alt-S.

The shortcut event occurs when the shortcut key combination is pressed in the appropriate context. The default context is when the widget is a child of the parent window, but this can be adjusted through the method `setShortcutContext`.

Checkable actions, and action groups Actions can be set checkable through the `setCheckable` method. When in a checked state, the `checked` property is `TRUE`. When a checkable action is checked the `toggled` signal is emitted, the argument `checked` passes in the state.

A `QActionGroup` can be used to group together checkable actions so that only one is checkable (like radio buttons). To use this, you create an instance and use the `addAction` to link the actions to the group.

Example 11.5: Creating an action

To create an action, say to "Save" an object requires a few steps. It is recommended that the main constructor is passed the parent widget the action will apply within.

```
parent <- Qt$QMainWindow()
saveAction <- Qt$QAction("Save", parent)
```

We could also pass the icon to the constructor, but instead set the icon, a shortcut, a tooltip, and a statusbar tip through the action's methods.

```
saveAction$setShortcut(Qt$QKeySequence("Save"))
iconFile <- system.file("images/save.gif", package="gWidgets")
saveAction$setIcon(Qt$QIcon(iconFile))
saveAction$setToolTip("Save the object")
saveAction$setStatusTip("Save the current object")
```

The action encapsulates a command, in this case we have a stub:

```
qconnect(saveAction, "triggered", function(checked)
  print("Save object"))
```

Menubars

Main windows may have a menubar. This may appear at the top of the window, or the menubar area on Mac OS X. Menubars are instances of `QMenuBar` which provides access to list of top-level submenus.

These submenus are added through `addMenu`, where a string with a possible shortcut are specied to label the menu. A `QMenu` instance is returned. To submenus one can add

1. nested submenus through the `addMenu` method,
2. an action through the `addAction` method, or
3. a separator through `addSeparator`.

Actions may be removed from a window through `removeAction`, but usually menu items are just disabled if their command is not applicable.

Example 11.6: Menu items

In a data editor application, one might imagine a menu item for coercion of a chosen column from one type to the next. In the following, we assume we have a function `colType` that returns the column type of the selected column or NA if no column is selected. We begin by making a menu bar, and a "Data" menu item. To this we add a few actions, and then a "Coerce" submenu. In the submenu, we use an action group so that only one type can be checked at a time. Actions must be added to both the action group and the submenu.

```
mb <- Qt$QMenuBar() # or parent$menuBar()
menu <- mb$addMenu("Data") # a submenu
#
```



```

menu$addAction(a <- Qt$QAction("Apply Function...", parent))
qconnect(a, "triggered", function() cat("apply ..."))
menu$addAction(a <- Qt$QAction("Relevel Factors...", parent))
qconnect(a, "triggered", function() cat("relevel ..."))
#
menu$addSeparator()
#
cmenu <- menu$addMenu("Coerce")
aList <- sapply(c("character", "factor", "numeric"),
               function(i) {
                 a <- Qt$QAction(i, parent)
                 a$setCheckable(TRUE)
                 qconnect(a, "toggled", function(checked) print(i)) ## stub
                 a
               })
actionGroup <- Qt$QActionGroup(w)
sapply(aList, function(i) actionGroup$addAction(i))
sapply(aList, function(i) cmenu$addAction(i))

```

In the application, we might include logic to update the menu items along the lines of the following. If no column type is available (no column is selected) we disable the submenu, otherwise we set the check accordingly. Of course, in the application we would ensure that checking the menu item updates the state in the data model through the triggered handler.

```

updateMenus <- function() {
  val <- colType()
  cmenu$setEnabled(!is.na(val))
  if(!is.na(val)) {
    aList[[val]]$setChecked(TRUE)
  }
}

```

Context menus

Context menus can be added to widgets using the same `QMenu` widget (not `QMenuBar`). The `popup` method will cause the menu to popup, but it needs to be told where. (The `exec` method will also popup a menu, but blocks other input.) The location of the popup is specified in terms of global screen coordinates, but typically the location known is in terms of the widgets coordinates. (For example, the point (0,0) being the upper-left corner of the widget.) The method `mapToGlobal` will convert for you. Position is in terms of a `QPoint` instance, which can be constructed or may be returned by an event handler. We illustrate both in the example.

Initiating the popup menu can be done in different ways. In the example below, we first show how to do it when a button is pressed. More natural ways are to respond to right mouse clicks, say. These events may be found

within event handlers, say the `mousePressEvent` event. (The `QMouseEvent` object passed in has a `button` method that can be checked.) However, the operating system may provide other means to initiate a popup. Rather than program these, Qt provides the `contextMenuEvent`. We can override that in a subclass, as illustrated in the example.

Example 11.7: Popup menus

We imagine a desire to popup possible function names that complete a string. Such suggestions are computed from a function in the `utils` package. We first show how to offer these in a popup menu we do this for a button press (not the most natural case):

```
b <- Qt$QPushButton("Completion example")
qconnect(b, "pressed", function(...) {
  ## compute popup
  popup <- Qt$QMenu()
  comps <- utils::matchAvailableTopics("mean")
  l <- sapply(comps, function(i) {
    a <- Qt$QAction(i, b)
    popup$addAction(a)
  })
  popup$popup(b$mapToGlobal(Qt$QPoint(0,0)))
})
```

More naturally, we might want this menu to popup on a right mouse click in a line edit widget. To implement that, we define a subclass and reimplement the `contextMenuEvent` method. We use the `globalPos` method of the passed through event to get the appropriate position.

```
qsetClass("popupmenuexample", Qt$QLineEdit)
#
qsetMethod("contextMenuEvent", popupmenuexample, function(e) {
  popup <- Qt$QMenu()
  comps <- utils::matchAvailableTopics(this$text)
  if(length(comps) > 10)
    comps <- comps[1:10] # trim if large
  sapply(comps, function(i) {
    a <- Qt$QAction(i, this)
    qconnect(a, "triggered", function(...) this$setText(i))
    popup$addAction(a)
  })
  popup$popup(e$globalPos())
})
e <- popupmenuexample()
```

Toolbars

Toolbars, giving easier access to a collection of actions or even widgets, can be readily added to a main window. The basic toolbar is an instance of the `QToolBar` class. Toolbars are added to the main window through the `addToolBar` method.

To add items to a toolbar we have:

1. `addAction` to add an action,
2. `addWidget` to embed a widget into the toolbar,
3. `addSeparator` to place a separator between items.

Actions can be removed through the `removeAction` method, although typically they are simply disabled as appropriate. The method actions will return a list of actions.

Toolbars can have a few styles. The orientation can be horizontal (the default) or vertical. The `setOrientation` method adjusts this with values specified by `Qt::Qt$Horizontal` or `Qt::Qt$Vertical`. The toolbuttons can show a combination of text and/or icons. This is specified through the method `setToolButtonStyle` with values taken from the `Qt::ToolButtonStyle` enumeration. The default is icon only, but one could use, say, `Qt::Qt$ToolButtonTextUnderIcon`.

Example 11.8: A simple toolbar

The following illustrates how to put in toolbar items to open and save a file. We suppose we have a function `getIcon` that returns a `QIcon` instance from a string.

We define a top-level window to hold our toolbar and be the parent for our actions that will be placed in the toolbar. We store them in a list for ease in manipulation at a later time in the program.

```
w <- Qt$QMainWindow()
a <- list()
a$open <- Qt$QAction("Open", w)
a$open$setIcon(getIcon("open"))
a$save <- Qt$QAction("Save", w)
a$save$setIcon(getIcon("save"))
```

We define our toolbar, set its button style and then add to top-level window in the next few commands.

```
tb <- Qt$QToolBar()
tb$setToolButtonStyle(Qt$Qt$ToolButtonTextUnderIcon)
w$addToolBar(tb)
```

Finally, we add the actions to the toolbar.

```
sapply(a, function(i) tb$addAction(i))
```

Statusbars

Main windows have room for a statusbar at the bottom of the window. The status bar is used to show programmed messages as well as any status tips assigned to actions.

A statusbar is an instance of the `QStatusBar` class. One may be added to a main window through the `setStatusBar` method. For some operating systems, a size grip is optional and its presence can be adjusted through the `sizeGripEnabled` property.

Messages may be placed in the status bar of three types: *temporary* where the message stays briefly, such as for status tips; *normal* where the message stays, but may be hidden by temporary messages, and *permanent* where the message is never hidden. In addition to messages, one can embed widgets into the statusbar.

The `showMessage` method places a temporary message. The duration can be set by specifying a time in milliseconds for a second argument. Otherwise, the message can be removed through `clearMessage`.

Use `addWidget` with a label to make a normal message, use `addPermanentWidget` to make a permanent message.

Dockable widgets

In Qt main windows have dockable areas where one can anchor widgets that can be easily detached from the main window to float if the user desires. An example use might be a toolbar or in a large GUI, a place to dock a workspace browser. The main methods are `addDockWidget` and `removeDockWidget`. Adding a dock widget requires first creating an instance of `QDockWidget` and then setting the desired widget through the dock widget's `setWidget` method. Widgets may go on any side of the central widget. The position is specified through the `DockWidgetArea` enumeration, with values such as `Qt::LeftDockWidgetArea`.

Dock widgets can be stacked or arranged in a notebook like manner. The latter is done by the `tabifyDockWidget`, which moves the second argument (a dock widget) on top of the first with tabs, like a notebook, for the user to select the widget.

Floating a dock widget is initiated by the user through clicking an icon in the widget's title bar or programatically through the `floating` property.

Example 11.9: Using a main widget for the layout of an IDE

This example shows how to mock up a main window (Figure ??) similar to the one presented by the web application, R-Studio. (rstudio.org).

We begin by setting a minimum size and a title for the main window.

```
w <- Qt$QMainWindow()  
w$setMinimumSize(800, 500)
```

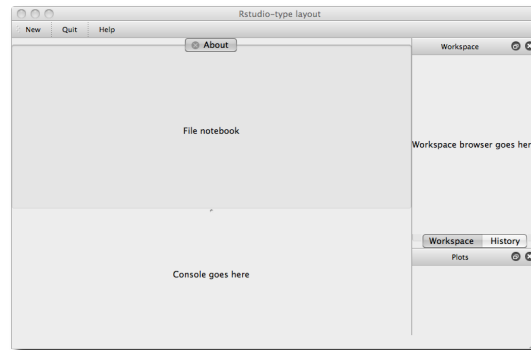


Figure 11.3: A mock of for a possible GUI using dockable widgets and a QMainWindow instance

```
w$setWindowTitle("Rstudio-type layout")
```

We add in a menu bar and toolbar. The actions are minimal, not including icons, commands etc. We show the file menu definitions.

```
l <- list()
mb <- Qt$QMenuBar()
w$setMenuBar(mb)
fmenu <- mb$addMenu("File")
fmenu$addAction(l$new <- Qt$QAction("New", w))
fmenu$addSeparator()
fmenu$addAction(l$open <- Qt$QAction("Open", w))
fmenu$addAction(l$save <- Qt$QAction("Save", w))
fmenu$addSeparator()
fmenu$addAction(l$quit <- Qt$QAction("Quit", w))
```

The toolbar has just a few actions added.

```
tb <- Qt$QToolBar()
w$addToolBar(tb)
tb$addAction(l$new)
tb$addSeparator()
tb$addAction(l$quit)
tb$addSeparator()
tb$addAction(l$help <- Qt$QAction("Help", w))
```

Our central widget holds two main areas: one for editing files and one for a console. As we may want to edit multiple files, we use a tab widget for that. A QSplitter is used to divide the space between the two main widgets.

```
centralWidget <- Qt$QSplitter()
centralWidget$setOrientation(Qt$Qt$Vertical)
w$setCentralWidget(centralWidget)
```

```
fileNotebook <- Qt$QTabWidget()
l <- Qt$QLabel("File notebook")
l$setAlignment(Qt$Qt$AlignCenter)
fileNotebook$addTab(l, "About")
fileNotebook$setTabsClosable(TRUE)
qconnect(fileNotebook, "tabCloseRequested", function(ind, nb) {
  nb$removeTab(ind)
}, user.data=fileNotebook)
centralWidget$addWidget(fileNotebook)
```

Our console widget is just a stub.

```
consoleWidget <- Qt$QLabel("Console goes here")
consoleWidget$setAlignment(Qt$Qt$AlignCenter)
centralWidget$addWidget(consoleWidget)
```

On the right side of the layout we will put in various tools for interacting with the R session. We place these into dock widgets, in case the user would like to place them elsewhere on the screen. Defining dock widgets is straightforward. We show a stub for a workspace browser.

```
workspaceBrowser <- Qt$QLabel("Workspace browser goes here")
wbDockWidget <- Qt$QDockWidget("Workspace")
wbDockWidget$setWidget(workspaceBrowser)
```

The workspace and history browser are placed in a notebook to conserve space. We add the workspace browser on the right side, then tabify the history browser (whose construction is not shown).

```
w$addDockWidget(Qt$Qt$RightDockWidgetArea, wbDockWidget)
w$tabifyDockWidget(wbDockWidget, hbDockWidget)
```

We next place a notebook to hold any graphics produced in a dock widget. This one occupies its own space.

```
plotNotebook <- Qt$QTabWidget()
pnDockWidget <- Qt$QDockWidget("Plots")
w$addDockWidget(Qt$Qt$RightDockWidgetArea, pnDockWidget)
```

Finally, we make status bar and add a transient message.

```
sb <- Qt$QStatusBar()
w$setStatusBar(sb)
sb$showMessage("Mock-up layout for an IDE", 2000)
```

Widgets

This chapter covers some of the basic dialogs and widgets provided by Qt, saving for later a discussion on widgets that have a model backend. Together with layouts, these form the basis for most user interfaces.

12.1 Dialogs

Qt implements the conventional high-level dialogs, including those for printing, selecting files, selecting colors, and, most usefully, sending simple messages and input requests to the user. We first introduce message and input dialogs. This is followed by a discussion of the infrastructure in Qt for implementing custom dialogs. Finally, we briefly introduce some of the remaining high-level dialogs, such as the file selector.

Message Dialogs

All dialogs in Qt are derived from `QDialog`. The message dialog, `QMessageBox`, communicates a textual message to the user. At the bottom of the dialog are a set of buttons, each representing a possible response. Normally, the type of message is indicated by an icon. If extra details are available, the dialog provides the option for the user to view them.

Qt provide two ways to create a message box. The simplest approach is to call a static convenience method for issuing common types of messages, including warnings and simple questions. The alternative, described later, involves several steps and offers more control at a cost of convenience. Here we take the simple path for presenting a warning dialog:

```
response <- Qt::QMessageBox::warning(parent = NULL, title = "Warning!",  
                                     text = "Warning message...")
```

This blocks the flow of the program until the user responds and returns the standard identifier for the button that was clicked. Each type of button corresponds to a fixed type of response. The standard button/response codes are listed in the `QMessageBox::StandardButton` enumeration. In this case, there

is only a single button, "QMessageBox\$Ok". The dialog is *modal*, meaning that the user cannot interact with the "parent" window until responding. If the "parent" is "NULL", as in this case, input to all windows is blocked. The dialog is automatically positioned near its parent, and if the parent is destroyed, the dialog is destroyed, as well. Additional arguments specify the available buttons/responses and the default response. We have relied on the default values for these.

For more control over the appearance and behavior of the dialog, we will take a more gradual path. First, we construct an instance of `QMessageBox`. It is possible to specify several properties at construction. Here is how one might construct a warning dialog:

```
dlg <- Qt$QMessageBox(icon = Qt$QMessageBox$Warning,
                      title = "Warning!",
                      text = "Warning text...",
                      buttons = Qt$QMessageBox$Ok,
                      parent = NULL)
```

This call introduces the `icon` property, which is a code from the `QMessageBox::Icon` enumeration and identifies a standard, themeable icon. The icon also implies the message type, just as a button implies a response type. We also need to specify the possible responses with the "buttons" argument.

Our dialog is already sufficiently complete to be displayed. However, we have the opportunity to specify further properties. Two of the most useful are `informativeText` and `detailedText`:

```
dlg$informativeText <- "Less important warning information"
dlg$detailedText <- "Extra details most do not care to see"
```

Both provide additional textual information at an increasing level of detail. The `informativeTextQMessageBox` will be rendered as secondary to the actual message text. To display the `detailedText`, the user will need to interact with a control in the dialog. An example is a stack trace for the warning.

After specifying the desired properties, the dialog is shown. The approach to showing the dialog depends on whether the dialog should be modal. A modal dialog is displayed with the `exec` method.

```
dlg$exec()
```

```
[1] 1024
```

As its name implies, `exec` executes a loop that will block until the user responds. As with the static convenience methods, the return value indicates the button/response.

To present a non-modal dialog, we first need to register a response listener, as the response will arrive asynchronously. Then we show, raise and activate the dialog:


```

qconnect(dlg, "finished", function(response) {
  ## handle response
  ## dlg$close() necessary?
})
dlg$show()
dlg$raise()
dlg$activateWindow()

```

There are several signals that indicate user response, including "finished", "accepted", and "rejected". The most general is "finished", which passes the button/response code as its only argument.

Modal dialogs may be window modal (`QtQtWindowModal`), where the dialog blocks all access to its ancestor windows, or application modal (`QtQtApplicationModal`) (the default) where all windows are blocked. To specify the type of modality, call `setWindowModality`.

To summarize, we present a general message box, supporting multiple responses:

```

dlg <- Qt$QMessageBox()
dlg$windowTitle <- "[This space for rent]"
dlg$text <- "This is the main text"
dlg$informativeText <- "This should give extra info"
dlg$detailedText <- "And this provides\neven more detail"
dlg$icon <- Qt$QMessageBox$Critical
dlg$standardButtons <- Qt$QMessageBox$Cancel | Qt$QMessageBox$Ok
## 'Cancel' instead of 'Ok' is the default
dlg$setDefaultButton(Qt$QMessageBox$Cancel)
if(dlg$exec() == Qt$QMessageBox$Ok)
  print("A Ok")

```

Input dialogs

The `QInputDialog` class provides a convenient means to gather information from the user and is in a sense the inverse of `QMessageBox`. Possible input modes include selecting a value from a list or entering text or numbers. By default, input dialogs consist of an input control, an icon, and two buttons: "Ok" and "Cancel".

Like `QMessageBox`, one can display a `QInputDialog` either by calling a static convenience method or by constructing an instance and configuring it before showing it. We demonstrate the former approach for a dialog that requests textual input:

```

text <- Qt$QInputDialog$getText(parent = NULL,
                                title = "Gather text",
                                label = "Enter some text")

```

The return value is the entered string, or "NULL" if the user cancelled the dialog. Additional parameters allow one to specify the initial text and to override the input mode, e.g., for password-style input.

We can also display a dialog for integer input. Here, we ask the user for an even integer between 1 and 10:

```
num <- Qt$QInputDialog$getInt(parent = NULL, title = "Gather integer",
                              label = "Enter an integer from 1 to 10",
                              value = 0, min = 2, max = 10, step = 2)
```

The number is chosen using a bounded spin box. To request a real value, call `Qt$QInputDialog$getDouble` instead.

The final type of input is selecting an option from a list of choices:

```
item <- Qt$QInputDialog$getItem(parent = NULL, title = "Select item",
                                 label = "Select a letter",
                                 items = LETTERS, current = 17)
```

The dialog contains a combo box filled with the capital letters. The initial choice is 0-based index 17, or the letter "R". The chosen string is returned.

`QInputDialog` has a number of options that cannot be specified via one of the static convenience methods. These option flags are listed in the `QInputDialog$InputDialogOption` enumeration and include hiding the "Ok" and "Cancel" buttons and selecting an item with a list widget instead of a combo box. If such control is necessary, we must explicitly construct a dialog instance, configure it, execute it and retrieve the selected item.

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Select item")
dlg$setLabelText("Select a letter")
dlg$setComboBoxItems(LETTERS)
dlg$setOptions(Qt$QInputDialog$UseListViewForComboBoxItems)
```

```
if (dlg$exec())
  print(dlg$textValue())
```

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

QDialog

Every dialog in Qt inherits from `QDialog`, and we can leverage it for our own custom dialogs.

What makes a dialog different is when it is modal. That is the `exec` call is used. This state may be broken by quitting the window through the window manager, or by calling the `done` method with an integer specifying the return value (say `Qt$QDialog$Accepted` or `Qt$QDialog$Rejected`). This closes the window and returns control. If the `QtQtWA_DeleteOnClose` attribute is set, the dialog will be deleted, otherwise it may be reused.

A simple example follows. We assume some parent window is defined as `parent`

Our dialog is defined and we adjust our window modality accordingly. Under Mac OS X, this will appear as a drop down sheet of the parent, not in a separate window.

```
dlg <- Qt$QDialog(parent)
dlg$setWindowModality(Qt$Qt$WindowModal)
dlg$setWindowTitle("A simple dialog")
```

Our dialog here is just a basic mock up. We use a horizontal box for the buttons, but in an real application would use the `QDialogButtonBox`

```
dlg$setLayout(lyt <- Qt$QVBoxLayout())
lyt$addWidget(Qt$QLabel("Layout dialog components here"))
blyt <- Qt$QHBoxLayout() # for buttons
lyt$addLayout(blyt)
```

Our buttons have callbacks that call the `done` method with a return value. We use `user.data` to pass in the dialog reference.

```
ok <- Qt$QPushButton("Ok")
cancel <- Qt$QPushButton("Cancel")
blyt$addWidget(ok)
blyt$addWidget(cancel)
qconnect(ok, "pressed", function(dlg) dlg$done(1), user.data=dlg)
qconnect(cancel, "pressed", function(dlg) dlg$done(0), user.data=dlg)

if(dlg$exec())
  print("Yes")
```

File and Directory choosing dialogs

`QFileDialog` allows the user to select files and directories, by default using the platform native file dialog. As with other dialogs there are static methods to create dialogs with standard options. These are `"getOpenFileName"`, `"getOpenFileNames"`, `"getExistingDirectory"`, and `"getSaveFileName"`. To select a file name to open we would have:

```
fname <- Qt$QFileDialog$getOpenFileName(NULL, "Open a file...", getwd())
```

All take as initial arguments a parent, a caption and a directory. Other arguments allow one to set a filter, say. For basic use, these are nearly as easy to use as R's `file.choose`. If a file is selected, `fname` will contain the full path to the file, otherwise it will be `NULL`.

To allow multiple selection, call the plural form of the method:

```
fnames <- Qt$QFileDialog$getOpenFileNames(NULL, "Open file(s)...", getwd())
```

To select a file name for saving, we have

12. WIDGETS

```
fname <- Qt$QFileDialog$getSaveFileName(NULL, "Save as...", getwd())
```

And to choose a directory,

```
dirname <- Qt$QFileDialog$getExistingDirectory(NULL, "Select directory", getwd())
```

To specify a filter by file extension, we use a name filter. A name filter is of the form `Description (*.ext *.ext2)` (no comma) where this would match files with extensions `ext` or `ext2`. Multiple filters can be used by separating them with two semicolons. For example, this would be a natural filter for R users:

```
rfilter <- paste("R files (*.R *.RData)",  
               "Sweave files (*.Rnw)",  
               "All files (*.*)", sep=";;")  
fnames <- Qt$QFileDialog$getOpenFileNames(NULL,  
                                           "Open file(s)...", getwd(),  
                                           rfilter)
```

Explicitly constructing a dialog Although the static functions provide most of the functionality, to fully configure a dialog, it may be necessary to explicitly construct and manipulate a dialog instance. Examples of options not available from the static methods are history (previously selected file names), sidebar shortcut URLs, and filters based on low-level file attributes like permissions.

Example 12.1: File dialogs

We construct a dialog for opening an R-related file, using the file names selected above as the history:

```
dlg <- Qt$QFileDialog(NULL, "Choose an R file", getwd(), rfilter)  
dlg$fileMode <- Qt$QFileDialog$ExistingFiles  
dlg$setHistory(fnames)
```

The dialog is executed like any other. To get the specified files, call `selectedFiles`:

```
if(dlg$exec())  
  print(dlg$selectedFiles())
```

12.2 Labels

As seen in previous example, basic labels in Qt are instances of the `QLabel` class. Labels in Qt are the primary means for displaying static text and images. Textual labels are the most common, and the constructor accepts a string for the text, which can be plain text or, for rich text, HTML. Here we use HTML to display red text:

```
1 <- Qt$QLabel("<font color='red'>Red</font>")
```

The class, by default, guesses if the string is rich text or not and in the above identifies the HTML. One can explicitly set the text format (`setTextFormat`) if need be.

The label text is stored in the `text` property. Properties relevant to text layout include: `alignment`, `indent` (in pixels), `margin`, and `wordWrap`.

12.3 Buttons

Buttons are created by `QPushButton` which inherits most of its functionality from `QAbstractButton`. A button has a `text` property for storing a label and an `icon` property to show an accompanying image.

Buttons are associated with commands. One may bind to the inherited signal `clicked`, which is called when the button is activated by the mouse, a short cut key, or a call to `clicked`. The callback receives a logical value checked if the button is “checkable.” Otherwise, `pressed` and `released` signals are emitted.

Button boxes

Dialogs often have a standard button placement that varies among operating systems. Qt provides the `QDialogButtonBox` class to store buttons. This class accepts the standard buttons listed in the `QDialogButtonBox::StandardButton` enumeration. Each standard button has a default role from a list of roles specified in `QDialogButtonBox::ButtonRole`, if a non-standard button is desired, then a role must be specified. The `addButton` method is used to add a button, as specified by either a standard button or by a label and role. This method returns a `QPushButton` instance.

In a dialog, a button may be designated as the default button for a dialog. To specify a default, the button’s `setDefault` is called with a value of `TRUE`.

To get feedback, one can connect each button to the desired signal. More conveniently, the button box has signals `accepted`, which is emitted when a button with the `AcceptRole` or `YesRole` is clicked; `rejected`, which is emitted when a button with the `RejectRole` or `NoRole` is clicked; `helpRequested`; or `clicked` when any button is clicked. For this last signal, the callback is passed in the button object.

Example 12.2: A yes-no-help set of buttons

We use a dialog button box to hold a standard set of buttons. Figure 12.1 shows the difference in their display for two different operating systems. Below, we just illustrate how to specify callbacks based on the button’s role, but only put in stubs for their commands.

```
db <- Qt$QDialogButtonBox()
```

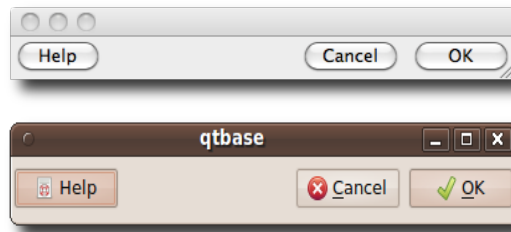


Figure 12.1: Dialog button boxes and their implementation under Mac OS X and Linux.

```
ok <- db$addButton(Qt$QDialogButtonBox$Ok)
db$addButton(Qt$QDialogButtonBox$Cancel)
## Or: db$addButton("Cancel", Qt$QDialogButtonBox$RejectRole)
db$addButton(Qt$QDialogButtonBox$Help)
#
ok$setDefault(TRUE)
#
qconnect(db, "accepted", function() print("accepted"))
qconnect(db, "rejected", function() print("rejected"))
qconnect(db, "helpRequested", function() print("help"))
```

Icons and pixmaps

In Figure 12.1 we see that in some toolkits buttons may carry a standard icon. A user can also pass in an icon through the `setIcon` method. Icons are instances of the `QIcon` class. Icons may be read in from files or from `QPixmap` instances. Although one may, it is not necessary to define icons for different sizes and display modes (normal, disabled, active, selected), as Qt can do so automatically.

For example, we define an icon and add it to a button as follows. The call to `setEnabled` shows how Qt can draw the icon in a disabled state.

```
ok <- system.file("images/ok.gif", package="gWidgets")
icon <- Qt$QIcon(ok)
b <- Qt$QPushButton("Ok")
b$setIcon(icon)
b$setEnabled(FALSE)
```

The `QPixmap` class creates a means to represent an image on a paint device. One can create icons from pixmaps, and the `pixmap` method will return a pixmap from an icon. (The size may be specified with integers, and the mode can be set). Pixmaps can be directly made through the constructor. For example, the following creates a button icon using code from the `ggplot` package.

```
png(f <- tempfile())
grid.newpage()
grid.draw(GeomHistogram$icon())
dev.off()
pix <- Qt$QPixmap(f)
b <- Qt$QPushButton("Histogram")
b$setIcon(Qt$QIcon(pix))
```

The QPixmap class has several methods for manipulating the image not discussed here.

12.4 Checkboxes

The QCheckBox class implements a checkbox. Like the QPushButton class, class inherits methods and properties from the QAbstractButton class. For example, the checkbox label is associated with the text property.

Qt has three states for a checkbox: the obvious Qt\$Qt\$Checked, Qt\$Qt\$Unchecked states, and a third indeterminate state specified by Qt\$Qt\$PartiallyChecked. These are stored in the checkState for setting and getting.

In addition to the inherited signals clicked, pressed, and released, The signal stateChanged is emitted. Callbacks are passed the state as an integer.

Groups of checkboxes

The QButtonGroup can be used to group together buttons, such as checkboxes, into logical units. The layout of the buttons is not managed by this widget. By default, the buttons are exclusive, like a radio button group, but this can be adjusted by passing FALSE to the method setExclusive. Buttons are added to the button group one-by-one through the method addButton. An optional ID can be passed in to identify the buttons, but it may be more convenient to work with the list of buttons returned by the buttons method.

When a button is clicked, pressed or released a signal is emitted, for example buttonClicked is when a button is clicked. The callback receives the button object.

Example 12.3: Using checkboxes to provide a filter

Filtering a data set by the levels of a factor is a familiar interface. This is commonly implemented by using a checkbox group with each level assigned to a toggle. Below we show how to do so for a cylinders variable, such is in the Cars93 data set of the MASS package.

We begin by defining the levels and a widget to store our buttons.

```
data(Cars93, package="MASS")
cyls = levels(Cars93$Cylinders)
w <- Qt$QGroupBox("Cylinders:")
lyt <- Qt$QVBoxLayout()
```

12. WIDGETS

```
w$setLayout(lyt)
```

We want to be able to select more than one button, so set the exclusive property accordingly.

```
bg <- Qt$QButtonGroup()
bg$setExclusive(FALSE)
```

Next we prepare a button for each value of `cyls`, add it to the layout and then the button group. Finally we initialize the buttons to all be checked.

```
sapply(seq_along(cyls), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cyls[i]))
  lyt$addWidget(button)
  bg$addButton(button, i)
})
sapply(bg$buttons(), function(i) i$setChecked(TRUE))
```

Our simple callback to the `buttonClicked` signal shows how to see which buttons were checked after the button was pressed. (The `buttonPressed` is called before the widget state reflects the button press.)

```
qconnect(bg, "buttonClicked", function(button) {
  checked <- sapply(bg$buttons(), function(i) i$checked)
  if(any(checked)) {
    ind <- Cars93$Cylinders %in% cyls[checked]
    print(sprintf("You've selected %d cases", sum(ind)))
  }
})
```

12.5 Radio groups

Radio buttons are created by the `QRadioButton` constructor, which inherits from `QAbstractButton`. Radio buttons are typically exclusive. To group buttons together, Qt links all buttons that share the same parent widget. Radio buttons, like other buttons, have a text and icon property. In addition, radio buttons have a checked or unchecked state which can be queried with `isChecked` and set with `setChecked`.

Example 12.4: Using a radio group to filter

Instead of a group of checkboxes, we might also filter through the exclusive selection offered by radio buttons.

First, we place the radio buttons into a list for easy manipulation. A `QButtonGroup` could also be used here, but we use an R-based approach for variety.

```
w <- Qt$QGroupBox("Weight:")
```



```
l <- list(Qt$QRadioButton("Weight < 3000", w),
         Qt$QRadioButton("3000 <= Weight < 4000", w),
         Qt$QRadioButton("4000 <= Weight", w)
        )
```

Next we define a layout for our radio button widgets. The buttons are children of `w` below, so are exclusive within that.

```
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

To add the widgets to the layout and set the first one to be checked we have the following:

```
sapply(l, function(i) lyt$addWidget(i))
l[[1]]$setChecked(TRUE)
```

The toggled signal is emitted twice when a button is clicked: once for when the check button is clicked, and once for the toggle of the previously checked button. Below, we condition on the value of checked to restrict to one call.

```
sapply(l, function(i) {
  qconnect(i, "toggled", function(checked) {
    if(checked) {
      ind <- which(sapply(l, function(i) i$isChecked()))
      print(sprintf("You checked %s.", l[[ind]]$text))
    }
  })
})
```

12.6 Sliders and spin boxes

Sliders and spin boxes are similar widgets used for selecting from a range of values. Sliders give the illusion of selecting from a continuum, whereas spinboxes offer a discrete choice, but underlying each is a selection from an arithmetic sequence. In Qt the two have similar method names.

Sliders

Sliders are implemented by `QSlider` a subclass of `QAbstractSlider`, which also provides functionality for scrollbars. Sliders in Qt are for selecting from integer values. To specify the range of possible values to select from the methods `setMinimum` and `setMaximum` are used (assuming integer values). Movement between the possible values is adjusted by `setSinglePageStep` with broader motion by `setPageStep`. (If we think of the arguments from, to, by of `seq` then these are the minimum, maximum and the single page step.) The actual value stored in the widget is found in the property value.

Sliders can be horizontal or vertical, the orientation can be changed by passing a `Qt::Orientation` value to the `setOrientation` method. To adjust the appearance to ticks for a slider, the method `setTickPosition` is used with values drawn from the `QSlider::TickPosition` enumeration (for example, `Qt$QSlider$TicksBelow`, `"TicksLeft"` or the default `"NoTicks"`). The method `setTickInterval` is used to specify an interval between the ticks.

The signal `valueChanged` is emitted when the slider is moved. It passes back the current value to a callback. The `sliderMoved` signal is similar, only the slider must be down, as when being dragged by a mouse.

Spin boxes

Spin boxes are derived from `QAbstractSpinBox` which provides the base class for `QSpinBox` (for integers), `QDoubleSpinBox` and `QDateTimeEdit` (not pursued here).

The methods have similar names as for sliders: `setMinimum`, `setMaximum`, and `setValue` have similar usages. The step size is provided by `setSingleStep`. The property wrapping can be set to `TRUE` to have the values wrap at the ends.

In Qt spinbuttons can have a prefix or suffix with the numbers. These are set by `setPrefix` or `setSuffix`.

As with sliders, the signal `valueChanged` is emitted when the spin button is changed,

Example 12.5: A range selector

We combine a slider and spinbox to make a range selection widget, offering the speedier movement of the slider, with the finer adjustments of the spin box.

Our slider construction sets values one-by-one.

```
sl <- Qt$QSlider()
sl$setMinimum(0)
sl$setMaximum(100)
sl$setSingleStep(1)
sl$setPageStep(5)
```

To style our slider we make it horizontal, set the tick position and interval through the following:

```
sl$setOrientation(Qt$Qt$Horizontal)
sl$setTickPosition(Qt$QSlider$TicksBelow)
sl$setTickInterval(10)
```

The basic spin box construction is relatively straightforward. We add a `"%"` suffix to make it seem like we are selecting a percent.

```
sp <- Qt$QSpinBox()
sp$setMinimum(0)
sp$setMaximum(100)
```

```
sp$setSingleStep(1)
sp$setSuffix("%")
```

To link the two widgets, we define callbacks for their `valueChanged` signal updating the other widget when there is a change. This could possibly cause an infinite loop, so we check if the suggested value is not equal to the current one before updating.

```
f <- function(value, obj) {
  if(! isTRUE(all.equal(value, obj$value)))
    obj$setValue(value)
}
qconnect(sp, "valueChanged", f, user.data=sp)
qconnect(sl, "valueChanged", f, user.data=sl)
```

12.7 Single-line text

As seen in previous examples, a widget for entering or displaying a single line of text is provided by the `QLineEdit` class. The `text` property holds the current value. The text may be set as the first argument to the constructor, or through the method `setText` (provided `readOnly` is `FALSE`). Text may be inserted through the `insert` method, replacing the currently selected text or inserting at the cursor. One can programatically position the cursor by index through the `setCursorPosition` method. As is typical, the index is 0 for the left most position, 1 for between the first and second character, etc. The right-most index can be found from `nchar(widget$text)`, say. The `setSelection` method takes two indices to indicate the left and right bounds of the selection.

When there is a selection, the methods `hasSelectedText` and `selectedText` are applicable. If `dragEnabled` is `TRUE` the selected text may be dragged and dropped on the appropriate targets.

If desired, it is possible to mask the displayed text with asterisks (common with passwords) by setting the `echoMode` property. Value are taken from the `QLineEdit::EchoMode` enumeration, e.g., `Qt::QLineEdit::Password`. If desired, the property `displayText` holds the displayed text.

Undo, redo The widget keeps an undo/redo stack. The methods `modified`, `isRedoAvailable`, `isUndoAvailable` are helpful in tracking if the text has changed and undo and redo can go through the changes.

The widget emits several different signals that are of use. The `cursorPositionChanged` signal is emitted as the cursor is moved. The old and new positions are passed along. The `selectionChanged` signal is emitted as the selection is updated. The `textChanged` signal is emitted when the text is changed. Any callback is passed the new text. Similarly for `textEdited`, the difference being that this signal is not emitted when text is set by `setText`. The distinction

between the signals `editingFinished` and `returnPressed` is due to the former being called only if a valid entry is given.

Completion

Using Qt's `QCompleter` framework, a list of possible words can be presented for completion. The word list is generally specified by a model, but may also be specified as a character vector to the constructor. A simple usage is presented by example.

Example 12.6: Using completion on the Qt object

The Qt environment has many components. This example shows how completion can assist in exploring them by name. We use a form layout to arrange our two line edit widgets – one to gather a class name and one for method and property names.

```
w <- Qt$QWidget()
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Class name", c_name <- Qt$QLineEdit())
lyt$addRow("Method name", m_name <- Qt$QLineEdit())
```

The completer for the class is constructed just one. We use `ls` to list the components of the environment. We see that completions are set for a line edit widget through the `setCompleter`.

```
c_comp <- Qt$QCompleter(ls(Qt))
c_name$setCompleter(c_comp)
```

The completion for the methods depends on the class. As such, we update the completion when editing is finished for the class name.

```
qconnect(c_name, "editingFinished", function() {
  cl <- c_name$text
  val <- get(cl, envir=Qt)
  if(!is.null(val)) {
    m_comp <- Qt$QCompleter(ls(val))
    m_name$setCompleter(m_comp)
  }
})
```

Masks and Validation

`QLineEdit` has various means to restrict and validate user input. The `maxLength` property can be set to restrict the number of allowed characters. To set a pattern for the possible answer, an input mask can be set through `setInputMask`. Input masks are specified through a string indicating a pattern. For example,

"999-99-9999" is for a U.S. Social Security number. The API for `QLineEdit` contains a full description.

As illustrated in Example 10.1, Qt also implements a validation framework where the value in the widget is validated before being committed. When a validator is set, using `setValidator` the method is called before the value is transferred from the GUI to the widget. The function can return one of three different states of validity "Acceptable" (i.e, `Qt::QValidator::Acceptable`), "Invalid", or the indeterminate "Intermediate". The function is passed the current value and the index of the cursor.

The validator must be an instance of a subclass of `QValidator` with a `validate` method. This requires constructing a subclass.

12.8 Multi-line text

Multi-line text is displayed and edited through the `QTextEdit` class. This widget allows for more than plain text. It may show rich text through HTML syntax including images, lists and tables.

As a basic text editor, the widget is a view for an underlying `QTextDocument` instance. The document can be used to retrieve that document, and `setDocument` to replace it. The method `toPlainText` is used to retrieve the text as plain text with a corresponding `setPlainText` for replacing the text. Text can also be added. The `append` method will append the text to the end of the document, `insertPlainText` will insert the text at the current cursor position replacing any selection, and `paste` will paste the current clipboard contents into the current cursor position. (Use copy and cut to replace the clipboards contents.) More complicated insertions can be handled through the `QTextCursor` class.

To illustrate

```
te <- Qt::QTextEdit()
te$setPlainText("The quick brown fox")
te$append("jumped over the lazy dog")
```

```
te$toPlainText()
```

```
[1] "The quick brown fox\njumped over the lazy dog"
```

Formatting By default, the widget will wrap text as entered. For use as a code editor, this is not desirable. The `setLineWrapMode` takes values from the enumeration `QTextEdit::LineWrapMode`, to control this. A value `Qt::QTextEdit::NoWrap` will turn off wrapping. When wrapping is enabled, one can control how with the enumeration `QTextOption::WrapMode` and the method `setWordWrap`.

The `setAlignment` method aligns the current paragraph with values from `Qt::Alignment`.

Formatting is managed by the `QTextFormat` class with subclasses such as `QTextCharFormat`, for formatting at the character level and `QTextBlockFormat` to format blocks of text. (A document is comprised of blocks containing paragraphs etc.) There are methods to adjust the alignment, font properties, margins etc.

The text cursor The cursor position is returned from the `position` method of the text cursor, which is found through the `textCursor` method. The position is an integer indicating the index of the cursor if the buffer is thought of as a single string. To set the cursor position one first gets the text cursor, sets its position with `setPosition`, then sets the textedit's text cursor through `setTextCursor`.

For example, to move the cursor to the end can be done with

```
n <- nchar(te$toPlainText())
cursor <- te$textCursor()
cursor$setPosition(n)
te$setTextCursor(cursor)
```

Selections The text cursor `hasSelection` method indicates if a selection exists. If it does, the method `selection` returns a `QTextDocumentFragment` which also has a method `toPlainText`. A selection is determined by the position of the cursor and an anchor. The latter is found through `anchor`. However, to move the anchor is a bit more difficult. Basically, as the name suggests, you keep an anchor in place and move the position. The `movePosition` method's second argument is one of `Qt::QTextCursor::KeepAnchor` or `Qt::QTextCursor::MoveAnchor`. When the anchor is kept in place and the cursor moved a selection is defined. The movement of the position can be specified through the enumeration `QTextCursor::MoveOperation` with values like "Start", "End", "NextWord", etc.

To set the selection to include the first three words of the text, we have:

```
cursor <- te$textCursor()
cursor$movePosition(Qt::QTextCursor::Start) ## default is move anchor
cursor$movePosition(Qt::QTextCursor::WordRight, Qt::QTextCursor::KeepAnchor, 3)
te$setTextCursor(cursor)
```

Note the extra space at the end here:

```
te$textCursor()$selection()$toPlainText()
```

```
[1] "The quick brown "
```

Navigation The widget has a `find` method to move the selection to the next instance of a string. The enumeration `QTextDocument::FindFlag` can modify the search with values "FindBackward", "FindCaseSensitively" and "FindWholeWords".

For example, we can search through a standard typesetting string starting at the cursor point for the common word “qui” as follows:

```
te <- Qt$QTextEdit(LoremIpsum)          # some text
te$find("qui", Qt$QTextDocument$FindWholeWords)
```

```
[1] TRUE
```

```
te$textCursor()$selection()$toPlainText()
```

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

Signals There are several useful signals that are emitted by the widget. Some deal with changes: `cursorPositionChanged`, `selectionChanged`, `textChanged`, and `currentCharFormatChanged`. Others allow one to easily update any actions to reflect the state of the widget. These include `copyAvailable`, which passes in a boolean indicating “yes”; `redoAvailable`, which passes a boolean indicating availability; and similarly `undoAvailable`.

To get keystroke information, one can create a subclass and implement the `keyPressEvent` method, say.

Example 12.7: A tabbed text editor

This example shows how to combine the text edit with a notebook widget to create a widget for editing more than one file at a time.

We begin by defining a sub-class of `QTextEdit` so that we can define a few useful properties and method. In this example, we limit ourselves to just a few customizations of the edit widget.

```
qsetClass("TextEditSheet", Qt$QTextEdit, function(parent=NULL) {
  super(parent)

  this$nb <- NULL          # text edit notebook
  this$filename <- NULL    # name of file, tab

  setLineWrapMode(Qt$QTextEdit$NoWrap)
  setFontFamily("Courier")
})
```

The `nb` property stores the parent notebook widget allowing us to look this up when we have the sheet object.

```
qsetMethod("setNb", TextEditSheet, function(nb) this$nb <- nb)
```

The file name is used for saving the sheet and for labeling the notebook tab. We define a getter and setter. The setter is also tasked with updating the tab label and illustrates how the `nb` property is employed for this.

```
qsetMethod("filename", TextEditSheet, function() filename)
qsetMethod("setFilename", TextEditSheet, function(fname) {
  this$filename <- fname
  ## update tab label
  notebook <- this$nb$notebook
  ind <- notebook$indexOf(this)
  notebook$setTabText(ind, basename(fname))
})
```

Next, we define a few methods for the sheet. First, one to save the file. We use the filename property for the suggested name to save as.

```
qsetMethod("saveSheet", TextEditSheet, function() {
  fname <- Qt$QFileDialog$getSaveFileName(this,
                                           "Save file as...", filename())
  if(!is.null(fname)) {
    txt <- this$toPlainText()
    writeLines(strsplit(txt, "\n")[[1]], con=fname)
  }
})
```

We will see soon, that a method to test if a given sheet is the currently visible sheet is useful. This method returns a logical by comparing the current index of the notebook with the index of the sheet.

```
qsetMethod("isCurrentSheet", TextEditSheet, function() {
  notebook <- this$nb$notebook
  notebook$currentIndex == notebook$indexOf(this)
})
```

Finally we want to initialize a new sheet. This involves defining callbacks that update the actions in the parent notebook as appropriate. Later we define a minimal set of actions for this example and store them in the `tbactions` property of the parent notebook.

```
qsetMethod("initSheet", TextEditSheet, function() {

  qconnect(this, "redoAvailable", function(yes) {
    if(isCurrentSheet())
      this$nb$tbactions$redo$setEnabled(yes)
  })
  qconnect(this, "undoAvailable", function(yes) {
    if(isCurrentSheet())
      this$nb$tbactions$undo$setEnabled(yes)
  })
  qconnect(this, "selectionChanged", function() {
    hasSelection <- this$textCursor()$hasSelection()
    if(isCurrentSheet())
      sapply(c("cut", "copy"), function(i)
        this$nb$tbactions[[i]]$setEnabled(hasSelection))
  })
})
```



```

    })
    qconnect(this, "textChanged", function() {
        if(isCurrentSheet()) {
            mod <- this$document()$isModified()
            this$nb$tbactions$save$setEnabled(mod)
        }
    })
})

```

Next, we turn to the task of defining a notebook container. In fact we subclass `QMainWindow` so that we can add a toolbar. The notebook is then a property, as are the actions. Our constructor customizes the notebook, sets up the actions and toolbar, then opens with a blank sheet.

```

qsetClass("TextEditNotebook", Qt$QMainWindow, function(parent=NULL) {
    super(parent)

    ## properties
    this$tbactions <- list()
    this$notebook <- Qt$QTabWidget()

    notebook$setTabsClosable(TRUE)
    notebook$setUsesScrollButtons(TRUE)
    qconnect(notebook, "tabCloseRequested",
              function(ind) notebook$removeTab(ind))
    qconnect(this$notebook, "currentChanged", function(ind) {
        if(ind > 0)
            updateActions()
    })
    setCentralWidget(notebook)

    initActions()
    makeToolbar()
    newSheet()
})

```

For our example, we implement a basic set of methods. This one maps down from the parent notebook widget to a sheet.

```

qsetMethod("currentSheet", TextEditNotebook, function() {
    this$notebook$currentWidget()
})

```

This method is used to open a new sheet, either a blank one or some initial text.

```

qsetMethod("newSheet", TextEditNotebook,
           function(title="..Scratch..", str=NULL) {

                a <- TextEditSheet()           # a new sheet

```

12. WIDGETS

```
a$setNb(this)           # set parent notebook
a$initSheet()           # initialize the sheet

this$notebook$addTab(a, "") # add to the notebook
ind <- this$notebook$indexOf(a)
this$notebook$setCurrentIndex(ind)

if(!is.null(str))        # set text if present
  a$setPlainText(str)
a$setFilename(title)     # also updates tab
})
```

To open a file in a new sheet, we use a standard dialog to get the filename to open, then call `newSheet`.

```
qsetMethod("openSheet", TextEditNotebook, function() {
  dlg <- Qt$QFileDialog(this, "Select a file...", getwd())
  dlg$setFileMode(Qt$QFileDialog$ExistingFile)
  if(dlg$exec()) {
    fname <- dlg$selectedFiles()
    txt <- paste(readLines(fname), collapse="\n")
    newSheet(fname, txt)
  }
})
```

We have several actions possible in our GUI, such as the standard cut, copy and paste. We define them for the parent notebook, but the actions primarily work at the sheet level. The `initActions` is called to set up the actions. We don't show the entire method, as it is repetitive, but is primarily of this type:

```
actions <- list()           # hold the actions
## make an action for open. This is TextEditNotebook instance
actions$open <- Qt$QAction("open", this)
actions$open$setShortcut(Qt$QKeySequence("Open"))
qconnect(actions$open, "triggered", function(obj) obj$openSheet(), user.data=this)
## ... etc. ...
this$tbactions <- actions
```

Whenever a new sheet is shown, the state of the actions should reflect that sheet. The `updateActions` method does this task. The constructor has a callback for this method whenever the sheet changes.

```
qsetMethod("updateActions", TextEditNotebook, function() {
  cur <- currentSheet()
  if(is.null(cur))
    return()

  a <- this$tbactions
  a$redo$setEnabled(FALSE)
```

```

a$undo$setEnabled(FALSE)
a$cut$setEnabled(cur$textCursor()$hasSelection())
a$copy$setEnabled(cur$textCursor()$hasSelection())
a$paste$setEnabled(cur$canPaste())

})

```

Finally, the `makeToolbar` method, which is not shown, simply maps the actions to a toolbar. Our GUI might also benefit from a menu bar, an exercise left for the reader.

Context menus Context menus may be added to the text edit widget, provided you create a subclass. The following example shows how to add to the standard context menu, which is available by the `createStandardContextMenu` method. The key is to redefine the `contextMenuEvent` method.

```

qsetClass("QTextEditWithMenu", Qt$QTextEdit)
#
qsetMethod("contextMenuEvent", QTextEditWithMenu, function(e) {
  m <- this$createStandardContextMenu()
  if(this$textCursor()$hasSelection()) {
    curVal <- this$textCursor()$selection()$toPlainText()
    comps <- utils:::matchAvailableTopics(curVal)
    comps <- setdiff(comps, curVal)
    if(length(comps) > 0 && length(comps) < 10) {
      m$addSeparator() # add actions
      sapply(comps, function(i) {
        a <- Qt$QAction(i, this)
        qconnect(a, "triggered", function(checked) {
          this$insertPlainText(i)
        })
        m$addAction(a)
      })
    }
  }
  m$exec(e$globalPos())
})
te <- QTextEditWithMenu()

```


Widgets using the MVC framework

13.1 Model View Controller implementation in Qt

The model-view-controller architecture adds a complexity to widgets that is paid in order to create more flexible and efficient use of resources. Keeping the model separate from the view allows multiple views for the same data. It also allows views to be more responsive when there are large data sets involved. The basic MVC architecture has a controller to act as a go-between for a model and its views. In Qt, this is different, an item in a model has a delegate that allows the user to see and perhaps edit the item's data within a view. Although custom delegates can be defined, we are content here to use those provided by Qt.

We discuss in the following various views: comboboxes, list views, table views and tree views. These are all familiar. The primary difficulty with using backend models is the specification and interaction with the model. For the simplest usages, Qt provides a set of convenience classes where the items and views are coupled.

Models in Qt are derived from the `QAbstractItemModel` class, although there are standard sub-classes for list, table and hierarchical models (`QStringListModel`, `QAbstractListModel`, `QAbstractTableModel`, and `QStandardItemModel`). In addition, the `qtbase` package provides the `qdataFrameModel` constructor to provide an mapping from a data frame to an item model.

A model is comprised of items, each having an associated index. Items have associated data which may vary based on the role (Table 13.1) the item is playing. For example, one role describes the data for display whereas another role describes the data for editing. This allows, for example, a numeric item, to be displayed with just a few digits, but edited with all of them. Both descriptions, and others, are stored in the item's data and set through the `setData` method. For the `qdataFrameModel` constructor these roles are encoded as additional, specially named, columns in the data frame.

An item has several properties that are described by the `Qt::ItemFlag` enumeration. These indicate whether an item is selectable (`ItemIsSelectable`),

Table 13.1: Partial list of roles that an item can hold data for and the class of the data.

Constant	Description
DisplayRole	How data is displayed (QString)
EditRole	Data for editing (QString)
ToolTipRole	Displayed in tooltip (QString)
StatusTipRole	Displayed in status bar (QString)
SizeHintRole	Size hint for views (QSize)
DecorationRole	(QColor, QIcon, QPixmap)
FontRole	Font for default delegate (QFont)
TextAlignmentRole	Alignment for default delegate (Qt::AlignmentFlag)
BackgroundRole	Background for default delegate (QBrush)
ForegroundRole	Foreground for default delegate (QBrush)
CheckStateRole	Indicates checked state of item (Qt::CheckState)

editable (ItemIsEditable), enabled (ItemIsEnabled), checkable (ItemIsCheckable) etc. The flags returns the flag recording combinations of these.

13.2 The item-view classes

The use of a model separate from a view can be relatively complicated. As such, Qt provides classes that couple a standard model and view. Defining the model then is much easier. In the case of a combobox and a list view the model can be specified simply through a character vector.

Comboboxes

The `QComboBox` class implements the combobox widget. The `QListWidget` class implements a basic list view. Both do similar things – allow selection from a list of possible values, although differently. As such, their underlying methods are mostly similar although their implementations have a different class inheritance. We discuss here the basic usage for comboboxes. Both widgets are views for an underlying model and we restrict ourselves, for now, to the simplest choice of these models.

By basic usage we mean setting a set of values for the user to choose from, and for comboboxes optionally allowing them to type in a new value. In the simplest use, we can define the items in their model through a character vector through the method `addItem`s. (The items must be character, so numeric vectors must be coerced. This works through a conversion into a `QStringList` object.) One can remove items by index with `removeItem`. To allow a user to enter a value not in this list, the property `editable` can be set to `TRUE`.

Once values are entered the property `currentIndex` holds the 0-based index of the selected value. This will be `-1` when no selection is made, and may be meaningless if the user can edit the values. Use `setCurrentIndex` to set the popup value by index (`-1` to leave unselected). The `count` method returns the number of items available.

The property `currentText` returns the current text. This will be a blank string if there is no current index. There is no corresponding `setCurrentText`, rather one can use the `findText` method to get the index of the specified string. There is an option to adjust how finding occurs through the enumeration `Qt::MatchFlags`.

The signal `activated` is emitted when the user chooses an item. When activated, the item index is passed to the callback. This signal is emitted when the user finishes editing of the text by the return key. The `highlighted` signal is emitted when the popup is engaged and the user mouses over an entry. For editable comboboxes, the signal `editTextChanged` is emitted after each change to the text.

Example 13.1: An example of one combobox updating another

This example shows how one combobox, to select a region in the U.S., is used to update another, which lists states in that region. We will use the following data frame for our data., which we split into a list.

```
df <- data.frame(name=state.name, region=state.region,
                 highlight=state.x77[,1], stringsAsFactors=FALSE)
l <- split(df, df$region)
```

Our region combobox is loaded with the state regions. When one is selected, the callback for `activated` will first remove any items in the state combobox, then add in the appropriate states. The `ind` index is used to determine which region.

```
region <- Qt$QComboBox()
state <- Qt$QComboBox()
region$addItem(names(l))
region$setCurrentIndex(-1) # no selection
qconnect(region, "activated", function(ind) {
  state$clear()
  state$addItem(l[[ind+1]]$name) # add
})
```

The state combobox shows how the `highlighted` signal can be employed. In this case, information about the highlighted state is placed in the window's title. Not really a great choice, but sufficient for this example.

```
qconnect(state, "highlighted", function(ind) {
  pop <- l[[region$currentText]][ind + 1, "highlight"]
  w$setWindowTitle(sprintf("Population is %s", pop))
})
```

Finally, we use a form layout to organize the widgets.

```
w <- Qt$QGroupBox("Two comboboxes")
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Region:", region)
lyt$addRow("State:", state)
```

A list widget

The `QListWidget` provides an easy-to-use widget for displaying a set of items for selection. It uses an item-based model for its data. The `QListView` widget provides a more general framework with different backend models. As with comboboxes, we can populate the items directly from a character vector through the `addItem` method. However, here we mostly focus on interacting with the widget through the item model.

The items in a `QListWidget` instance are of the `QListWidgetItem` class. New items can be constructed directly through the constructor. The first argument is the text and the optional second argument a parent `QListWidget`. If no parent is specified, the item may be added through the methods `addItem`, or `insertItem` where the row to insert is specified by index.

`QListWidget` items can have their text specified at construction or through the method `setText` and optionally have an icon set through the `setIcon`. There are also methods to set a status bar tip or a tooltip.

The method `takeItem` is used to remove items specified by their index, `clear` will remove all of them.

Once an item is added to list widget it can be referenced several ways. The currently selected item is returned by `currentItem`, whereas `currentRow` returns the current row by index, and `currentIndex` returns a `QModelIndex` instance (with a method `row` to get the index). As well, any item may be referenced by row index through `item` or position (say within an event handler) by `itemAt`. One can search for the items with the `findItems` method, which returns a list of items. An optional second argument uses the `Qt::MatchFlags` enumeration to adjust how matches are made, for example `Qt::MatchRegExp` to match by regular expression.

Selection By default, single selection mode is enabled. This can be adjusted through the `setSelectionMode` argument by specifying a value in `QAbstractItemView::SelectionMode`, such as `SingleSelection` or `ExtendedSelection`. Extended selection allows the user to extend the current selection by simultaneously pressing the control key or the shift key (selecting all items between the current selection and the newly selected item).

To retrieve the selected values, the method `selectedItems` will return the items in a list.

Setting an item to be selected is done through `setCurrentItem`. The first argument is the item, the optional second argument one of the `QItemSelectionModel::SelectionFlag` enumeration. If specified as `Qt::QItemSelectionModel::Select` (the default) the item will be selected, but other choices are possible such as "Deselect" or "Toggle".

Checkable items The underlying items may be checkable. This is initiated by setting an initial check state (`setCheckState`) with a value from "Checked" (`Qt::Qt$Checked`), "Unchecked" or "PartiallyChecked". For example, we can populate a list widget and set the values unchecked with.

```
w <- Qt$QListWidget()
w$addItem(state.name)
sapply(1:w$count, function(i)
  w$item(i-1)$setCheckState(Qt$Qt$Unchecked))
```

Then, after checking a few we can get the state along the lines of:

```
sapply(1:8, function(i) as.logical(w$item(i-1)$checkState()))
```

```
[1] TRUE FALSE TRUE TRUE FALSE FALSE TRUE TRUE
```

This uses the fact that the enumeration for "Unchecked" is 0 and "Checked" is 2.

Signals There are several signals that are emitted by the widget. Chief among them are `itemActivated`, which is emitted when a user clicks on an item or presses the activation key. The latter is what distinguishes it from the `itemClicked` signal. For capturing double clicks there is `itemDoubleClicked`. For these three, the underlying item is passed to the callback. The `itemSelectionChanged` signal is emitted when the underlying selection is changed.

Example 13.2: Filtering example

We illustrate the widget with a typical filtering example, where a user types in values to narrow down the available choices. We begin by setting up our widgets.

```
w <- Qt$QGroupBox("Filtering example")
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Filter:", f <- Qt$QLineEdit()) # define f
lyt$addRow("State:", lw <- Qt$QListWidget()) # define lw
lyt$addRow("", b <- Qt$QPushButton("Click me")) # define b
```

For convenience, we use a built-in data set for our choices. We populate the list widget and add a tooltip indicating the area.

```
for(i in state.name) {
```

13. WIDGETS USING THE MVC FRAMEWORK

```
item <- Qt$QListWidgetItem(i, lw)           # populate
txt <- sprintf("%s has %s square miles", i, state.x77[i, "Area"])
item$setToolTip(txt)
}
```

For this example, we allow multiple selection.

```
lw$setSelectionMode(Qt$QAbstractItemView$ExtendedSelection)
```

The following callback updates the displayed items so that only ones matching the typed in string are displayed. Rather than compare each item to the matched items, we simply hide them all then unhide those that match.

```
qconnect(f, "textChanged", function(str) {
  matching <- lw$findItems(str, Qt$Qt$MatchStartsWith)
  sapply(seq_len(lw$count), function(i) lw$item(i-1)$setHidden(TRUE))
  sapply(matching, function(i) i$setHidden(FALSE))
})
```

The following shows how we can grab the selected values.

```
qconnect(b, "pressed", function() {
  vals <- sapply(lw$selectedItems(), function(i) i$text())
  print(vals)
})
```

Example 13.3: Combining a combobox and list widget to select a variable name

This example shows how we can combine a combobox and a list widget to select a variable name from a data frame. Here we select a value by dragging it. As such we need to define a sub-class of `QListWidget` to implement the `mousePressEvent`.

```
qsetClass("DraggableListWidget", Qt$QListWidget,
  function(parent=NULL) {
    super(parent)
    this$df <- NULL
  })
```

The property `df` holds the name of the dataframe that will be selected through a combobox. Here is a method to set the value.

```
qsetMethod("setDf", DraggableListWidget,
  function(df) this$df <- df)
```

For drag and drop we show how to serialize an arbitrary R object to pass through to the drop target. We pass in a list of the data frame name and the selected variable name. The method `setData` takes a MIME type (which we arbitrarily define) and a value. This value will be retrieved by the `data` method and we can then call `unserialize`.

```
qsetMethod("mousePressEvent", DraggableListWidget, function(e) {
  item <- itemAt(e$pos())
  val <- list(df=this$df, var=item$text())

  md <- Qt$QMimeData()
  md$setData("R/serialized-data", serialize(val, NULL))

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)

  drag$exec()
})
```

With this, we know create a widget to hold the combobox and the list box. The constructor creates the widgets, lays them out, initializes the data sets then sets a handler to update the variable list when the dataframe selector does.

```
qsetClass("VariableSelector", Qt$QWidget, function(parent=NULL) {
  super(parent)

  this$dfcb <- Qt$QComboBox()
  this$varList <- DraggableListWidget()

  lyt <- Qt$QVBoxLayout()
  lyt$addWidget(dfcb)
  lyt$addWidget(varList)
  varList$setSizePolicy(Qt$QSizePolicy$Expanding,
                        Qt$QSizePolicy$Expanding)
  setLayout(lyt)

  updateDataSets()
  qconnect(dfcb, "activated", function(ind) {
    updateVarList(dfcb$currentText)
  })
})
```

Our method to update the data frame choice is a bit convoluted as we try to keep the currently selected data frame, if possible.

```
qsetMethod("updateDataSets", VariableSelector, function() {
  curVal <- this$dfcb$currentText
  this$dfcb$clear()
  x <- ls(envir=.GlobalEnv)
  dfs <- x[sapply(x, function(i)
    is.data.frame(get(i, inherits=TRUE)))]
  if(length(dfs)) {
    this$dfcb$addItem(dfs)
    if(is.null(curVal) || !curVal %in% dfs) {
```

```
        this$dfcb$setCurrentIndex(-1)
        updateVarList(NULL)
      } else {
        this$dfcb$setCurrentIndex(which(curVal == dfs))
        updateVarList(curVal)           # curVal NULL, or a name
      }
    }
  })
```

Finally, we need to update the list of variables to reflect the state of the combo box selection. Here we define a helper method to display an appropriate icon based on the class of the variable.

```
getIconFile <- function(x) UseMethod("getIconFile")
getIconFile.default <- function(x)
  Qt$QIcon(system.file("images/numeric.gif", package="gWidgets"))
getIconFile.factor <- function(x)
  Qt$QIcon(system.file("images/factor.gif", package="gWidgets"))
getIconFile.character <- function(x)
  Qt$QIcon(system.file("images/character.gif", package="gWidgets"))
```

This method populates the variable list to reflect the indicated data frame. As items are automatically drag enabled, we do not need to add anything more here, as we've implemented the `mousePressEvent`.

```
qsetMethod("updateVarList", VariableSelector, function(df=NULL) {
  this$varList$setDf(df)
  this$varList$clear()
  if(!is.null(df)) {
    d <- get(df)
    sapply(names(d), function(i) {
      item <- Qt$QListWidgetItem(i, this$varList)
      item$setIcon(getIconFile(d[,i]))
    })
  }
})
```

A table widget

The `QTableWidget` class provides a widget for displaying tabular data in an item-based approach, similar to `QListWidget`. The `QTableView` widget is more flexible, but also more demanding, as it has the ability to have different data models (which can be much faster with large tables). As such, only if your needs are not too complicated will this widget will be a good choice.

The dimensions of the table must be set prior to adding items. The methods `setRowCount` and `setColumnCount` are used.

The `QTableWidget` class has a built in model that is populated item by item. Items are of class `QTableWidgetItem` and are created first, then in-

serted into the widget, by row and column, through the method `setItem` (These operations are not vectorized and can be slow for large tables.) Items can be removed by row-and-column index with `takeItem`. The item can be reinserted. The `clear` method will remove all items, even headers items, whereas, `clearContents` will leave the headers. Both keep the dimensions.

As with `QListWidget`, items have various properties that can be adjusted. The text can be specified to the constructor, or set through `setText`. Text alignment is specified through `setTextAlignment`. The font may be configured through `setFont`. The methods `setBackground` and `setForeground` are used to adjust the colors.

Items may also have icons (`setIcon`), tooltips (`setToolTip`), and statusbar tips (`setStatusTip`).

Similar to `QListWidget`, `QTableWidget` instances are returned by the method `item`, with a specification of row and column; and by `itemAt`, only with a specification of a position. The `findItems` method will return a list of items matching a string. Also, there is the method `currentItem`, to return the currently selected item. From an item, its column and row can be found through its methods `column` and `row`.

Item flags As mentioned, items may have several different properties: are they editable, draggable, ...? To specify, one sets an items flags with values taken from the `Qt::ItemFlag` enumeration. The possible values are: `"NoItemFlags"`, `"ItemIsSelectable"`, `"ItemIsEditable"`, `"ItemIsDragEnabled"`, `"ItemIsDropEnabled"`, `"ItemIsUserCheckable"`, `"ItemIsEnabled"`, and `"ItemIsTriState"` (has three check states). To make an item checkable, one must first set the check state. By default, the widget is selectable, editable, drag and drop-pable, checkable and enabled. To remove a flag, one can specify all the ones they want, or use integer arithmetic and subtract. E.g., to remove the editable attribute one has this possibility:

```
item <- Qt$QTableWidgetItem("Set not editable")
if(item$flags() & Qt$Qt$ItemIsEditable)
  item$setFlags(item$flags() - Qt$Qt$ItemIsEditable)
```

Headers Columns may have headers (horizontal ones, rows have vertical headers). These are set all at once by specifying a character vector to `setHorizontalHeaderLabels`, or can be set with an item by `setHorizontalHeaderItem`. The header itself, of class `QHeaderView`, is returned by `horizontalHeader`. Headers have the method `setVisible` to toggle their visibility. To make the last column stretch to fill the available space is specified through the header vier method `setStretchLastSection` with a value of `TRUE`.

Otherwise, to specify the width of a column programmatically the method `setColumnWidth` is available. One specifies the column, then the width in pixels.

Sorting and Filtering This widget can have its rows sorted by the values of a column through the method `sortItems`. One specifies the column by index, and an order. The default is "AscendingOrder", the alternative is `QtQtDescendingOrder`. Sorting should be done after the table is populated with items.

Rows and columns can be hidden through the table widget's methods `setRowHidden` and `setColumnHidden`. This can be used for filtering purposes without redrawing the table.

Selection For the table widget one can easily select rows, columns, blocks and even combinations thereof. An underlying selection model implements selection, but the `QTableWidget` class provides an easier interface. The currently selected items are returned as a list through the method `selectedItems`.

A given cell may be selected by index (row then column) through the method `setCurrentCell`, or if the item instance is known by `setCurrentItem`. In addition, an optional selection flag from the enumeration `QItemSelectionModel::SelectionFlag` can be specified with values among "Select", to add the item to the selection; "Clear", to clear all selection; "Toggle", to toggle the specified item; "Rows", to extend the selection to the enclosing row; and similarly for "Columns". For an item itself, the method `setSelection` takes a logical value to indicate the selection state.

The current selection can be cleared by selecting an item with the "Clear" attribute, or by grabbing the underlying selection model and calling its `clearSelection` method. E.g, something like:

```
tbl$selectionModel()$clearSelection()
```

Signals The `QTableWidget` class has a number of signals it emits. They mostly come in pairs: "cell" ones passing in the row and column index and "item" ones passing in an item reference. For example, `cellClicked` and `itemClicked`, both called when a cell is clicked. Also there are `cellDoubleClicked`, `cellEntered`, `cellPressed`, `currentCellChanged` and `cellChanged` (with similar "item" ones). Also of interest is the `itemSelectionChanged` which is called when the selection changes.

Example 13.4: Selection of variables

This example shows how to use the table widget to select variable names in a data set. The `QListWidget` can also be used for this, but with the table widget we can add more detail in other columns. We begin by placing the table widget into a box layout.

```
tbl <- Qt$QTableWidget()
tbl$setSizePolicy(Qt$QSizePolicy$Expanding, Qt$QSizePolicy$Expanding)
```

```
lyt <- Qt$QVBoxLayout()
lyt$addWidget(tbl)
```

We use a `QDialogButtonBox` to hold our two buttons.

```
db <- Qt$QDialogButtonBox()
lyt$addWidget(db)
db$addButton(Qt$QDialogButtonBox$Ok)
db$addButton(Qt$QDialogButtonBox$Cancel)
qconnect(db, "accepted", function() runCommand())
qconnect(db, "rejected", function() w$hide())
```

We assume we have a data frame, `df`, with two columns – one for the variable name and one for detail, in this case the class of the variable.

We populate our table first by setting its dimensions and then setting the headers. We stretch the last column and set the column width of the first to accomodate the longest variable name in our example.

```
tbl$clear()
tbl$setRowCount(nrow(df))
tbl$setColumnCount(ncol(df))
tbl$setHorizontalHeaderLabels(c("Variable", "Class"))
tbl$horizontalHeader()$setStretchLastSection(TRUE)
tbl$setColumnWidth(0, 200)
tbl$verticalHeader()$setVisible(FALSE)
```

Each row in this example has two items, one for the checkable item holding the variable name and one for detail. Here we add row by row.

```
for(i in 1:nrow(df)) {
  item <- Qt$QTableWidgetItem(df[i,1])
  item$setCheckState(Qt$Qt$Unchecked)
  tbl$setItem(i-1, 0, item)

  item <- Qt$QTableWidgetItem(df[i,2])
  item$setTextAlignment(Qt$Qt$AlignLeft)
  tbl$setItem(i-1, 1, item)
}
```

Finally, this command is called when the "Ok" button is pressed. For simplicity we do two passes through the data to grab the text and the check state (which we must coerce to get a logical value).

```
runCommand <- function() {
  n <- tbl$rowCount
  x <- sapply(seq_len(n), function(i) tbl$item(i-1, 0)$text())
  ind <- sapply(seq_len(n), function(i) tbl$item(i-1, 0)$checkState())
  print(x[as.logical(ind)])
}
```

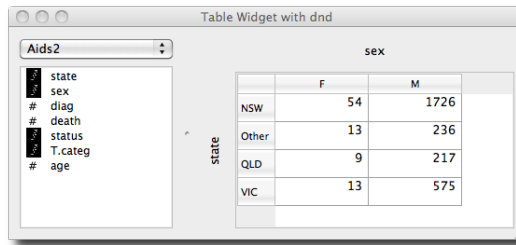


Figure 13.1: A table widget to display contingency tables and a means to specify the variables through drag and drop.

Example 13.5: A drag and drop interface to xtabs

This examples uses a table widget to display the output from `xtabs`. To specify the variable names, we use the `VariableSelector` class defined in Example 13.3. That provides a means to drag the variable. We next define a drop label, similar to that defined in Example ???. We show the major differences only. Our constructor is similar, but adds three properties: a function to call for the drop event, a value beyond the text, and an angle for text rotation. We define accessors for these properties, but do not show them here.

```
qsetClass("DropLabelRotation", Qt$QLabel, function(parent=NULL) {
  super(parent)

  this$dropFunction <- function(obj, data) {}
  this$value <- list(df=NULL, var=NULL)
  this$angle <- 0L

  setAcceptDrops(TRUE)
  setAlignment(Qt$Qt$AlignHCenter | Qt$Qt$AlignVCenter)
})
```

To handle the drag events we override the methods `dragEnterEvent`, `dragLeaveEvent`, and `dropEvent`. Only the latter is shown. For that we check for data with MIME type `R/serialized-data`. In that case, we call `unserialize` on the data and then the function in the `dropFunction` property.

```
qsetMethod("dropEvent", DropLabelRotation, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  md <- e$mimeType()
  if(md$hasFormat("R/serialized-data")) {
    data <- unserialize(md$data("R/serialized-data"))
    dropFunction(this, data)
  }
})
```



```

    setBackgroundRole(Qt::QPalette::Window)
    e$acceptProposedAction()
  }
})

```

For this application we want to be able to show text in the drop label in a rotated state. The following allows this. We override the `paintEvent` event if the `angle` property is non-zero, otherwise we call the parent class's `paintEvent`. The chapter on `qtpaint` will cover the techniques used in `drawRotatedText`. In short we translate the origin to the center of the label's rectangle, rotate coordinates by the angle, then place the text within this new rectangle/

```

qsetMethod("paintEvent", DropLabelRotation, function(e) {
  if(!this$angle) {
    super("paintEvent", e)
  } else {
    p <- Qt::QPainter()
    p$begin(this)
    this$drawRotatedText(p)
    p$end()
    update()
  }
})
qsetMethod("drawRotatedText", DropLabelRotation, function(p) {
  w <- this$width; h <- this$height
  p$save()
  p$translate(w/2, h/2)
  p$rotate(this$angle)
  p$drawText(qrect(-h, -w, h, w), Qt::Qt::AlignCenter, this$text)
  p$restore()
})

```

Our main widget consists of three widgets: two drop labels for the contingency table and a table widget to show the output. This could be extended to include a third variable for three-way tables, but we leave that exercise for the interested reader. The constructor simply calls two methods to be defined next.

```

qsetClass("XtabsWidget", Qt::QWidget, function(parent=NULL) {
  super(parent)

  initWidgets()
  initLayout()
})

```

Class 'R::GlobalEnv::XtabsWidget' with 318 public **methods**

We have three widgets to initialize. The `xlabel` is pretty basic: we define a widget and set a drop function. For the `ylabel` we also adjust the

rotation and constrain the width based on the font size. The latter is necessary, as otherwise the label width reflects the length of the dropped text. The `clearLabels` just sets the text and initializes the values for the labels. It is not shown.

```
qsetMethod("initWidgets", XtabsWidget, function() {  
  ## make Widgets  
  this$xlabel <- DropLabelRotation()  
  this$ylabel <- DropLabelRotation()  
  pt <- this$ylabel$font$pointSize()  
  this$ylabel$setMinimumWidth(2*pt); this$ylabel$setMaximumWidth(2*pt)  
  this$ylabel$setRotation(-90L)  
  
  this$tw <- Qt$QTableWidget()  
  
  clearLabels()  
  
  f <- function(obj, data) {  
    obj$setValue(data)  
    obj$setText(data$var)  
    this$makeTable()  
  }  
  this$xlabel$setDropFunction(f)  
  this$ylabel$setDropFunction(f)  
})
```

We don't show the layout code for this widget, it is a simple application of `QGridLayout`, but do show how we create the call to `xtabs` to create the data after the drop events.

```
qsetMethod("makeTable", XtabsWidget, function() {  
  df <- this$xlabel$getValue()$df  
  if(is.null(xVar <- this$xlabel$getValue()$var)) {  
    out <- NULL  
  } else if(is.null(yVar <- this$ylabel$getValue()$var)) {  
    f <- formula(sprintf("~ %s", xVar))  
    out <- xtabs(f, data=get(df))  
  } else {  
    f <- formula(sprintf("~ %s + %s", yVar, xVar))  
    out <- xtabs(f, data=get(df))  
  }  
  if(!is.null(out))  
    updateTableWidget(out)  
})
```

Finally, for the `XtabsWidget` class we define a method to update the table widget. We have two cases and the code repeats itself a bit. The basic tasks are to set the dimensions of the table, update the header labels, and then populate the table.

```

qsetMethod("updateTableWidget", XtabsWidget, function(out) {
  tw <- this$tw
  tw$clear()
  ndims <- length(dim(out))
  if(ndims == 1) {
    tw$setRowCount(1)
    tw$setColumnCount(dim(out))

    tw$setHorizontalHeaderLabels(names(out))
    tw$horizontalHeader()$setVisible(TRUE)
    tw$verticalHeader()$setVisible(FALSE)
    sapply(seq_along(out), function(i) {
      item <- Qt$QTableWidgetItem(as.character(out[i]))
      item$setTextAlignment(Qt$Qt$AlignRight)
      tw$setItem(0, i-1, item)
    })
  } else if(ndims == 2) {
    tw$setRowCount(dim(out)[1])
    tw$setColumnCount(dim(out)[2])

    tw$setHorizontalHeaderLabels(colnames(out))
    tw$setVerticalHeaderLabels(rownames(out))
    tw$horizontalHeader()$setVisible(TRUE)
    tw$verticalHeader()$setVisible(TRUE)

    for(i in 1:dim(out)[1])
      for(j in 1:dim(out)[2]) {
        item <- Qt$QTableWidgetItem(as.character(out[i,j]))
        item$setTextAlignment(Qt$Qt$AlignRight)
        tw$setItem(i-1, j-1, item)
      }
  }
})

```

Figure 13.1 shows the widget placed within a splitter window after a few variables have been dragged.

A Tree Widget

The `QTreeWidget` provides a tree view based on a coupled tree model. It is used in a manner similar to the table widget, however, one must adjust for the hierarchical nature of the data it can display. The widget shows a specific number of columns, which is specified via `setColumnCount`. The column headers are set through `setHeaderLabels` by specifying a vector of column names.

Tree Widget Items The widget organizes itself through items, with each item having 1 or more columns. The items in the tree widget are instances of the `QTreeWidgetItem` class. An item displays a value in each column. The method `setText` takes two arguments, an integer specifying a column, and a value to display. Similarly, other such item methods require one to specify the column. These methods include: `setTextAlignment`, `setFont`, `setToolTip`, `setStatusTip`, and `setIcon`. To retrieve these values, the method `data` takes a column number and a role, such as `Qt::DisplayRole` to get the text.

Heirarchy A tree is used to represent a heirarchy. The items have several methods related to this. Each item has a `parent` method pointing to the parent item, or `NULL` if a top-level item. (In the latter case, the tree widget's `invisibleRootItem` method returns the parent item.) The method `child` returns the child item, specified by index. The index of a given item in the parent is returned by `indexOfChild`. The total number of children for an item is returned by `childCount`. There is not a `nextSibling` method, but one could do something like:

```
nextSibling <- function(tr, i) {  
  parent <- i$parent()  
  if(is.null(parent))  
    parent <- tr$invisibleRootItem()  
  ind <- parent$indexOfChild(i)  
  n <- parent$childCount()  
  if(ind + 1 < n)  
    parent$child(ind + 1)           # 1 already added  
  else  
    NULL  
}
```

New child items are added with either with `addChild`, adding at the end; or `insertChild` adding the child item at the specified index. Child items are removed by the `removeChild` method or the `takeChild`, if specifying by index. The latter returns the item to be reparented if desired.

The view can be made to show if a child is present by calling `setChildIndicatorPolicy` with a value of `Qt::TreeWidgetItem::ShowIndicator` (or one of `DontShowIndicator` or `DontShowIndicatorWhenChildless`).

When an item has children and an indicator is shown, the view may be expanded by clicking on this indicator. The item method `isExpanded` returns `TRUE` when an item is expanded. This state can be set via `setExpanded`. The methods `setDisabled` and `setHidden` are also available to disable or hide an item.

Adding items to a tree The children of the invisible root item are the “top-level” items. These are added through `addTopLevelItem` or the `insertTopLevelItem`

methods, the latter requiring an index to specify where. The basic idea is that each top-level item is prepared, along with its hierarchy, then added through these methods.

Selection The currently selected item is returned by `currentItem` and may be specified by item through `setCurrentItem`. For multiple selection, see the discussion on selection models 13.3.

Signals The widget emits several signals, most passing the item and the column to the handler. Some useful signals are `itemClicked`, `itemDoubleClicked`, `itemExpanded`, and `itemActivated` (for clicking or the Enter key).

Example 13.6: A workspace browser

This example shows how to use the tree widget item to display a snapshot of the current workspace. Each object in the workspace maps to an item, where recursive objects with names will have their components represented in a hierarchical manner.

When representing objects in a workspace, we need to decide if an object has been changed. To do this, we use the `digest` function from the `digest` package to store a simple means to compare a current object with a past one. To store this information, we create a subclass of `QTreeWidgetItem` with a `digest` property.

```
require(digest)
qsetClass("OurTreeWidgetItem", Qt$QTreeWidgetItem, function(parent=NULL) {
  super(parent)
  this$digest <- NULL
})
qsetMethod("digest", OurTreeWidgetItem, function() digest)
qsetMethod("setDigest", OurTreeWidgetItem, function(value) {
  this$digest <- value
})
```

This subclass will be used for top-level items in our tree. We define the following to create an item from a variable name. Since we call this function recursively, we have an argument indicating of the item is a top-level item.

We could modify the display of each item to suit more sophisticated tastes, e.g., icons, but for this example stop by adding a column for the class of the object.

As we use the name of an object to display it, when checking for recursive structures, we also check that they have a `names` attribute.

```
makeItem <- function(varname, obj=NULL, toplevel=FALSE) {
  if(is.null(obj))
    obj <- get(varname, envir=.GlobalEnv)
```

```
if(toplevel) {
  item <- OurTreeWidgetItem()
  item$setDigest(digest(obj))
} else {
  item <- Qt$QTreeWidgetItem()
}

item$setText(0, varname)
item$setText(1, paste(class(obj), collapse=", "))
## icons, fonts children...

if(is.recursive(obj) && !is.null(attr(obj, "names"))) {
  item$setChildIndicatorPolicy(Qt$QTreeWidgetItem$ShowIndicator)
  for(i in names(obj)) {
    newItem <- makeItem(i, obj[[i]])
    item$addChild(newItem)
  }
}
item
}
```

Next we define a few simple functions to add, remove and replace a top-level item.

```
addItem <- function(tr, varname) {
  newItem <- makeItem(varname, toplevel=TRUE)
  tr$addTopLevelItem(newItem)
}
```

We can remove an item by index with the `takeTopLevelItem` method, but here it is more convenient to specify the item to remove.

```
removeItem <- function(tr, item) {
  root <- tr$invisibleRootItem()
  root$removeChild(item)
}
```

```
replaceItem <- function(tr, item, varname) {
  removeItem(tr, item)
  addItem(tr, varname)
}
```

Our main function is one that checks the current workspace and updates the values in the tree widget accordingly. This could be set on a timer to be called periodically, or called in response to user input.

As can be seen, we consider three cases: items no longer in the workspace to remove, new items to add, and finally items that may be new. Here is where we check the digest value to see if they need to be replaced. Rather

than fuss with inserting new items, we simply add them to the end and then sort the values.

```
updateTopLevelItems <- function(tr) {
  objs <- ls(envir=.GlobalEnv)

  curObjs <- lapply(seq_len(tr$topLevelItemCount), function(i) {
    item <- tr$topLevelItem(i-1)
    list(item=item, value=item$data(0, role=Qt$Qt$DisplayRole))
  })
  cur <- sapply(curObjs, function(i) i$value)
  names(curObjs) <- cur

  ## we have three types here:
  removeThese <- setdiff(cur, objs)
  maybeSame <- intersect(cur, objs)
  addThese <- setdiff(objs, cur)

  tr$setUpdatesEnabled(FALSE)
  for(i in removeThese) {
    item <- curObjs[[i]]$item
    removeItem(tr, item)
  }

  for(i in maybeSame) {
    obj <- get(i, envir=.GlobalEnv)
    if(digest(obj) != curObjs[[i]]$item$digest()) {
      replaceItem(tr, curObjs[[i]]$item, i)
    }
  }

  for(i in addThese)
    addItem(tr, i)

  tr$sortItems(0, Qt$Qt$AscendingOrder)
  tr$setUpdatesEnabled(TRUE)
}
```

Now we define some functions for traversing the hierarchical tree structure. First we define a function to find the index of an item, be it a top-level item or an ancestor of one.

```
indexFromItem <- function(tr, item) {
  parent <- item$parent()
  if(is.null(parent))
    parent <- tr$invisibleRootItem()
  parent$indexOfChild(item) + 1
}
```

We find it convenient to think in terms of a path rather than an item, either a vector of indices or names. The `index` argument below decides what to return. the basic approach here is to walk backwards by calling the parent method until it is null.

```
pathFromItem <- function(tr, item, index=TRUE) {
  getVal <- function(item)
    ifelse(index, indexFromItem(tr, item),
           item$data(0, role=0))

  path <- getVal(item)
  while(!is.null(item <- item$parent())) {
    path <- c(getVal(item), path)
  }
  path
}
```

This function reverses the last. It takes a path, specified by indices, and finds the item if possible.

```
itemFromPath <- function(tr, path) {
  item <- tr$invisibleRootItem()
  for(i in path) {
    item <- item$child(i)
    if(is.null(item))
      return(NULL)
  }
  item
}
```

As an illustration, we show how one can use `findItems` to return a list of items matching a query. In this case, we pass in a regular expression to match against the `class` column (column 1.)

```
matchClass <- function(tr, classname) {
  allItems <- sapply(seq_len(tr$topLevelItemCount), function(i) {
    tr$topLevelItem(i-1)
  })
  matched <- tr$findItems(classname, Qt$Qt$MatchRegExp, 1) # column 1
  sapply(allItems, function(i) i$setHidden(TRUE))
  sapply(matched, function(i) i$setHidden(FALSE))
}
```

Finally we show how this can be used. First, we set the column count and the header labels to match our `makeItem` function above.

```
tr <- Qt$QTreeWidget()
tr$setColumnCount(2) # name, class
tr$setHeaderLabels(c("Name", "Class"))
```

This call initializes the display.


```
updateTopLevelItems(tr)
```

An typical signal handler, to be called when an item is clicked, is illustrated next. Here we print both the path and a summary of the object in the workspace the path points to.

```
qconnect(tr, "itemClicked", function(item, column) {
  path <- pathFromItem(tr, item, index=FALSE)
  obj <- get(path[1], envir=.GlobalEnv)
  if(length(path) > 1)
    obj <- obj[[path[-1]]]          # get object
  str(obj)
  print(path)
  updateTopLevelItems(tr)
})
```

13.3 Item models and their views

In this section, we consider models and their views decoupled from each other. We focus our discussion to tabular data, although we note that Qt provides various means to provide models for hierarchical data.

Using a data frame for a model

For tabular data a model can be made quite easily with the constructor `qdataFrameModel` provided by `qtbases`. This function maps a data frame to the model, returning an instance of class `DataFrameModel` a subclass of `QAbstractTableModel`.

The default role is for display of the data. Other roles can be defined by adding additional columns with a specific naming convention. A column with name `a` can have its role information specified with another column with name `.a.ROLE` where `ROLE` is one of available roles, such as "decoration", "edit", "toolTip", "statusTip", or "textAlignment".

The following illustrates a basic usage. We use a `QTableView` instance for a view. The model is connected to the view through its `setModel` method.

Example 13.7: Using `qdataFrameModel` to list variables in a data frame

This example will show how to use `qdataFrameModel` to create a means to select a variable from a data frame. We will use a built-in data set for this.

```
nms <- names(Cars93)
d <- data.frame("Variables"=nms, stringsAsFactors=FALSE)
```

We will add an icon for decoration. To do so, we create a list of icons for each of the classes represented in the variables we have. Here we use some images from the `gWidgets` package.

```
icons <- sapply(c("factor", "numeric"), function(i) {  
  Qt$QPixmap(system.file(sprintf("images/%s.gif", i),  
                               package="gWidgets"))  
})  
icons$integer <- icons$numeric
```

To add the icons as a decoration, we simply assign to the appropriately named component of our data frame. Data frames can store such data, although the default print method does not work.

```
d$.Variables.decoration <- sapply(nms, function(i)  
  icons[class(Cars93[[i]])])
```

Similarly, we set an informative tooltip for each variable:

```
d$.Variables.toolTip <- sapply(nms, function(i) capture.output(str(Cars93[[i]])))
```

We now define a model and associate it with a view. In this example we use a `QListView` instance, although in most uses of this model we would use the `QTableView` class. We discuss views later in Section 13.3.

```
model <- qdataFrameModel(d)  
view <- Qt$QListView()  
view$setModel(model)
```

For the list view, we have a few properties to adjust, in this case we highlight alternating rows as follows:

```
view$setAlternatingRowColors(TRUE)
```

Table Models

The `qdataFrameModel` creates a subclass of `QAbstractTableModel`, itself a subclass of `QAbstractItemModel`. One can subclass `QAbstractTableModel` to provide a custom model for a view. This may be useful in different instances, for example, if the cell columns need to be computed dynamically or if editing of cells is desired.

Before discussing methods that must be implemented, we mention some useful inherited methods. For a table model, items are organized by row and column. This allows one to easily refer to an item by index. The inherited `index` method uses 0-based row and column arguments to return a `QModelIndex` instance. In turn, this index instance has methods `row` and `column` for getting back the coordinate information.

The `match` method can be used to return items in a column that match a specific string for a given role. The column is specified by a model index and the role by one of the `Qt::ItemDataRole` enumeration. The optional third argument is one of the `Qt::MatchFlag` enumeration, specifying how the matching is to occur. Matches are returned as a list of indices.

The sort method can be used to sort the data by the values in a column. The column is specified by its index, and the order is one of `Qt::AscendingOrder` or `Qt::DescendingOrder`.

Required methods The basic implementation of a subclass must provide the methods `rowCount`, `columnCount`, and `data`. The first two describe the size of the table for any views, the third describes to the view how to display data for a given role. The signature for the data method is `(index, role)`. The `index` value, an instance of `QModelIndex`, has methods `row` and `column`, whereas `role` is one of the roles defined in the `Qt::ItemDataRole` enumeration. This method should return the appropriate value for the given role. For example, if one is displaying numeric data, the `display` role might format the numeric values (showing a fixed number of digits say), yet the `edit` role might display all the digits so accuracy is not lost. If a role is not implemented, a value of `NULL` should be returned, as this is mapped to a null instance of `QVariant`.

One may also implement the `headerData` method to return the appropriate data to display in the header for a given role. The main one being `Qt::DisplayRole`.

Editable Models For editable models, one must implement the `flags` method to return a flag containing `ItemIsEditable` and the `setData` method. This has signature `(index, value, role)` where `value` is a character string containing the edited value. When a value is updated, one should call the `dataChanged` method to notify the views that a portion of the model is changed. This method takes two indices, together specifying a rectangle in the table.

To provide for resizable tables, Qt requires one to call some (of several such) functions so that any connected views can be notified. For example, an implemented `insertColumns` should call `beginInsertColumns` before adding the column to the model and then `endInsertColumns` just after.

Table views

A table model is typically displayed through the `QTableView` widget, although one can use the model – first column only – with `QListView` or even `QComboBox`. A custom view is also possible, as illustrated later. The table view widget inherits from `QAbstractItemView` which provides the method `setModel` to link the underlying model with a view. This link passes along information about the model being changed back to the view to process and connects the delegates that allow one to edit values in the view and have the changes propagate back to the model.

Once, connected, the table view can have its properties adjusted to control its appearance. For example, columns (similarly rows) may be hidden or

shown through the `setColumnHidden` method. Their widths can be adjusted by the mouse, or through `setColumnWidth`. The grid style can be adjusted through `setGridStyle`.

The view may allow sorting of the underlying model. This is enabled through `setSortingEnabled`. The method `sortByColumn` can be called specifying the column and a sort order (e.g. `Qt::AscendingOrder`).

Headers The table view has headers, horizontal ones for the columns and vertical ones for the rows. The methods `verticalHeader` and `horizontalHeader` return instances of the `QHeaderView` class. This class has many methods. We mention `setHidden`, to suppress the header display; `showSortIndicator`, to display a sort arrow; and `setStretchLastSection` to stretch the last section to fill the space. For tree views, this is `TRUE` for horizontal headers but not for table views.

Selection Models The type of selection possible is determined by the `selectionMode` of the view, the options for which are enumerated in `QAbstractItemView::SelectionMode`. Typical values are `SingleSelection` or `ExtendedSelection`, which allows one to extend their selection by pressing the Control or Shift keys while selecting.

For abstract item views, the underlying selection is handled by a selection model, which is found in the `selectionModel` property. A `QItemSelectionModel` instance can return the current selection by rows through `selectedRows` (with an argument to specify the column to check), columns (`selectedColumns`) or by index with `selectedIndexes`. In each case, a list of model indices is returned. The methods `row` and `column` are useful then. The selection model allows one to specify if the selection will apply to the entire row or column or if selection is cell by cell. This is done through the selection flags (`QItemSelectionModel::SelectionFlag`)

The selection can be updated programatically through the `setCurrentIndex` method. The index and a selection flag must be given. The latter enumerated in `QItemSelectionModel::SelectionFlag`, with values such as `Select`, to select the item; `Deselect`, `Clear`; `Toggle`; and `Rows` and `Columns` to select the entire range.

The selection model emits a few signals, notably `selectionChanged` is emitted when a new selection is made.

Example 13.8: Using a custom model to edit a data frame

This example shows how to create a custom model to edit a data frame. The performance is much less responsive than that provided by `qdataFrameModel`, as the bulk of the operations are done at the R level. We speed things up a bit, by placing column headers into the first row of the table, and not in the expected place. The numerous calls to a `headerData` method implemented

in R proved to be quite slow.

Our basic constructor simply creates a dataframe property and sets the data frame.

```
qsetClass("DfModel", Qt$QAbstractTableModel, function(df=data.frame(V1=character(0)), parent=
  super(parent)
  ## properties
  this$dataframe <- NULL

  setDf(df)
})
```

We need accessors for the dataframe property. When setting a new one, we call the `dataChanged` method to notify any views of a change.

```
qsetMethod("Df", DfModel, function() dataframe)
qsetMethod("setDf", DfModel, function(df) {
  this$dataframe <- df
  dataChanged(index(0, 0), index(nrow(df), ncol(df)))
})
```

All subclasses of `QAbstractTableModel` need to implement a few methods, here we do `rowCount` and `columnCount` using the dimensions of the current data frame.

```
qsetMethod("rowCount", DfModel, function(index) nrow(this$Df()) + 1)
qsetMethod("columnCount", DfModel, function(index) ncol(this$Df()))
```

The data method is the main method to implement. We wish to customize the data display based on the class of the variable represented in a column. We implement this with S3 methods. We define several. For example, to change the display role we have the following.

```
displayRole <- function(x, row, context) UseMethod("displayRole")
displayRole.default <- function(x, row, context)
  sprintf("%s", x[row])
displayRole.numeric <- function(x, row, context)
  sprintf("%.2f", x[row])
displayRole.integer <- function(x, row, context)
  sprintf("%d", x[row])
```

We see that numeric values are formatted to have 2 decimal points. By setting the display role, we can store numeric data with all its digits, so that we can edit the entire value, but have values formatted along the decimal point. The context argument is shown to indicate how to make the number of digits context sensitive.

Our data method has this basic structure (we avoid showing the cases for all the different roles).

```
qsetMethod("data", DfModel, function(index, role) {
```

```
d <- this$Df()
row <- index$row()
col <- index$column() + 1

if(role == Qt$Qt$DisplayRole) {
  if(row > 0)
    displayRole(d[,col], row)
  else
    names(d)[col]
} else if(role == Qt$Qt$EditRole) {
  if(row > 0)
    as.character(d[row, col])
  else
    names(d)[col]
} else {
  NULL
}
})
```

To allow the user to edit the values we need to override the `flags` method to return `ItemIsEditable` in the flag, so that any views are aware of this ability.

```
qsetMethod("flags", DfModel, function(index) {
  if(!index$isValid()) {
    return(Qt$Qt$ItemIsEnabled)
  } else {
    curFlags <- super("flags", index)
    return(curFlags | Qt$Qt$ItemIsEditable)
  }
})
```

To edit cells we need to implement a method to set data once edited. The trick is that value is a character, so we coerce to the right type for the column it is in. We do this with an S3 method. For example we have this (among others):

```
fitIn <- function(x, value) UseMethod("fitIn")
fitIn.default <- function(x, value) value
fitIn.numeric <- function(x, value) as.numeric(value)
```

The `setData` method is responsible for taking the value from the delegate and assigning it into the model, we use the `fitIn` function unless it is a column header.

```
qsetMethod("setData", DfModel, function(index, value, role) {

  if(index$isValid() && role == Qt$Qt$EditRole) {
    d <- this$Df()
    row <- index$row()
```

```

    col <- index$column() + 1

    if(row > 0) {
      x <- d[, col]
      d[row, col] <- fitIn(x, value)
    } else {
      names(d)[col] <- value
    }
    this$dataframe <- d
    dataChanged(index, index)

    return(TRUE)
  } else {
    super("setData", index, value, role)
  }
})

```

For a data frame editor, we may wish to extend the API for our table of items to be R specific. For example, this method allows one to replace a column of values.

```

qsetMethod("setColumn", DfModel, function(col, value) {
  ## pad with NA if needed
  n <- nrow(this$Df())
  if(length(value) < n)
    value <- c(value, rep(NA, n - length(value)))
  value <- value[1:n]
  d <- this$Df()
  d[,col] <- value
  this$dataframe <- d # only notify about this column
  dataChanged(index(0, col-1), index(rowCount()-1, col-1))
  return(TRUE)
})

```

We implement a method similar to the `insertColumn` method, but specific to our task. Since we may add a new column, we call the "begin" and "end" methods to notify any views.

```

qsetMethod("addColumn", DfModel, function(name, value) {
  d <- this$Df()
  if(name %in% names(d)) {
    return(setColumn(min(which(name == names(d))), value))
  }

  beginInsertColumns(Qt$QModelIndex(), columnCount(), columnCount())
  d <- this$Df()
  d[[name]] <- value
  this$dataframe <- d
  endInsertColumns()
}

```

```
    return(TRUE)
  })
```

To test this out, we define a model and set it in a view.

```
model <- DfModel(mtcars)
view <- Qt$QTableView()
view$setModel(model)
```

Finally, we customize the view by defining what triggers the delegate for editing and hide the row and column headers.

```
triggerFlag <- Qt$QAbstractItemView$DoubleClicked |
               Qt$QAbstractItemView$SelectedClicked |
               Qt$QAbstractItemView$EditKeyPressed
view$setEditTriggers(triggerFlag)
view$verticalHeader()$setHidden(TRUE)
view$horizontalHeader()$setHidden(TRUE)
```

Custom Views

One can write a custom view for a model. In the example, we show how to connect a label's text to the values in a column of a model. The base class for a view of an item model is `QAbstractItemView`. This class has many methods – that can be overridden – to cover keyboard, mouse, scrolling, selection etc. Here we focus on just enough for our example.

Example 13.9: A label that updates as a model is updated

This example shows how to create a simple custom view for a table model. We use the `DfModel` class developed in Example 13.8 to provide the model.

Our custom view class will be a sub-class of `QAbstractItemView`, although we don't consider the individual items below. Our goal is a simple illustration, where we provide a label with text summarizing the mean of the values in the first column of the model.

In the constructor we define a label property and call our `setModel` method.

```
qsetClass("CustomView", Qt$QAbstractItemView, function(parent=NULL) {
  super(parent)
  this$label <- Qt$QLabel()
  label$setMinimumSize(400,400)
  setModel(NULL)
})
```

The label has an accessor function which we use in our private `setLabelValue` method defined below. Here we get the model from the custom view.

```
qsetMethod("label", CustomView, function() label)
qsetMethod("setLabelValue", CustomView, function() {
  m <- this$model()
```



```
if(is.null(m)) {  
  txt <- "No model"  
} else {  
  df <- m$Df()  
  xbar <- mean(df[,1])  
  txt <- sprintf("The mean is %s", xbar)  
}  
label$setText(txt)  
},  
  access="private")
```

The main method call is when the data is changed. Here we update the label text when so notified.

```
qsetMethod("dataChanged", CustomView, function(upperIndex, lowerIndex) {  
  setLabelValue()  
})
```

To initialize the label text, we add a call to `setLabelValue` when the model is set.

```
qsetMethod("setModel", CustomView, function(model) {  
  super("setModel", model)  
  setLabelValue()  
})
```

Finally, to test this out, we put the view in a simple GUI along with an instance of an editable data frame. When we modify the model through that, the label's text updates accordingly.

```
model <- DfModel()  
model$setDf(mtcars)  
view <- Qt$QTableView()  
view$setModel(model)  
view$setEditTriggers(Qt$QAbstractItemView$DoubleClicked)  
cv <- CustomView()  
cv$setModel(model)  
w <- Qt$QWidget()  
lyt <- Qt$QHBoxLayout()  
w$setLayout(lyt)  
lyt$addWidget(view)  
lyt$addWidget(cv$label())
```


Qt paint

Tcl Tk: Overview

Tcl (“tool command language”) is a scripting language and interpreter of that language. Originally developed in the late 80s by John Ousterhout as a “glue” to combine two or more complicated applications together, it evolved overtime to find use not just as middleware, but also as a standalone development tool.

Tk¹ is an extension of Tcl that provides GUI components through Tcl. This was first developed in 1990, again by John Ousterhout. Tk quickly found widespread usage, as it made programming GUIs for X11 easier and faster. Over the years, other graphical toolkits have evolved and surpassed this one, but Tk still has numerous users.

Tk has a large number of bindings available for it, e.g. Perl, Python, Ruby, and through the `tcltk` package, R. The `tcltk` package was developed by Peter Dalgaard, and included in R from version 1.1.0. Since then, the package has been used in a number of GUI projects for R, most notably, the Rcmdr GUI.

Tk had a major change between version 8.4 and 8.5, with the latter introducing themed widgets. Many widgets were rewritten, and their API dramatically simplified. In `tcltk` there can be two different functions to construct a similar widget. For example, `tklabel` or `ttklabel`. The latter, with the `ttk` prefix, would be for the newer themed widget. We assume the Tk version is 8.5 or higher, as this was a major step forward. As of version 2.7.0, R for windows has been bundled with this Tk version, so there are no installation issues for that platform. As of writing, some linux distributions and Mac OS X still come with 8.4 which would need to be upgraded for the following.

¹ Tk has a well documented API (Tcl, a) (www.tcl.tk/man/tcl8.5). There are also several books to supplement. We consulted the one by Welch, Jones and Hobbs (Brent B. Welch, 2003) often in the development of this material. In addition, the Tk Tutorial of Mark Roseman (Tcl, b) (www.tkdcs.com/tutorial) provides much detail. R specific documentation include two excellent R News articles and a proceedings paper (Dalgaard, 2001), (?) and (Dalgaard) by Peter Dalgaard, the package author. A set of examples due to James Wettenhall (Wettenhall) are also quite instructive. A main use of `tcltk` is within the Rcmdr framework. Writing extensions for that is well documented in an R News article (Fox, 2007) by John Fox, the package author.

15.1 Interacting with Tcl

The basic syntax of Tcl is a bit unlike R. For example a simple string assignment would be made at `tclsh`, the Tcl shell with (using `%` as a prompt)

```
% set x {hello world}
hello world
```

Unlike R where braces are used to form blocks, this example shows how Tcl uses braces instead of quotes to group the words as a single string. The use of braces, instead of quotes, in this example is optional, but in general isn't, as expressions within braces are not evaluated.

The example above assigns to the variable `x` the value of `hello world`. Once assignment has been made, one can call commands on the value stored in `x` using the `$` prefix:

```
% puts $x
hello world
```

The `puts` command, in this usage, simply writes back its argument to the terminal. Had we used braces the argument would not have been substituted:

```
% puts {$x}
$x
```

More typical within the `tcltk` package is the idea of a subcommand. For example, the `string` command provides the subcommand `length` to return the number of characters in the string.

```
% string length $x
11
```

The `tcltk` package provides the low-level function `.Tcl` for direct access to the Tcl interpreter:

```
library(tcltk)
.Tcl("set x {some text}")           # assignment
```

```
<Tcl> some text
```

```
.Tcl("puts $x")                     # print
.Tcl("string length $x")            # call a command
```

```
<Tcl> 9
```

the `.Tcl` function simply sends a command as a text string to the Tcl interpreter and returns the result as an object of class `tclObj` (cf. `?Tcl`). These objects print with the leading `<Tcl>` (which we suppress here when there is no output). To coerce these values into characters, the `tclvalue`

function is used or the `as.character` function. They differ in how they treat spaces and new lines. Conversion to numeric values is also possible through `as.numeric`, but conversion to logical requires two steps.

The `.Tcl` function can be used to read in Tcl scripts as with `.Tcl("source filename")`. This can be used to run arbitrary Tcl scripts within an R session.

The Tk extensions to Tcl have a complicated command structure, and thankfully, `tcltk` provides some more conveniently named functions. To illustrate, the Tcl command to set the text value for a label object (`.label`) would look like

```
% .label configure -text "new text"
```

The `tcltk` provides a corresponding function `tkconfigure`. The above would be done as (assuming `l` is a label object):

```
tkconfigure(l, text="new text")
```

Although the Tcl statement appears to have the object oriented form of “object method arguments,” behind the scenes Tcl creates a command with the same name as the widget with `configure` as a subcommand. This is followed by options passed in using the form `-key value`. The Tk API for `ttklabel`’s `configure` subcommand is

pathName **configure** *?option? ?value option value ...?*

The *pathName* is the ID of the label widget. In the Tk documentation paired question marks indicate optional values. In this case, one can specify nothing, returning a list of all options; just an option, to query the configured value; the option with a value, to modify the option; and possibly do more than one at a time. For commands such as `configure`, if possible, there will correspond a function in R of the same name with a `tk` prefix, in this case `tkconfigure`. (The package `tcltk` was written before namespaces, so the “tk” prefix serves that role.) To make consulting the Tk manual pages easier in the text we would describe the `configure` subcommand as *ttklabel configure [options]*. (The R manual pages simply redirect you to the original Tk documentation, so understanding this is important for reading the API.) However, if such a function is present, we will use the R function equivalent when we illustrate code. Some subcommands have further subcommands. An example is to set the selection. In the R function, the second command is appended with a dot, as in `tkselection.set`. (There are just a few exceptions to this.)

The `tcl` function Within `tcltk`, the `tkconfigure` function is defined by

```
function(widget, ...) tcl(widget, "configure", ...)
```

The `tcl` function is the workhorse used to piece together Tcl commands. Behind the scenes it turns an R object, `widget`, into the *pathName* above (using

```
tcl(widget, subcommand, key=value, callback)
      /           |           |           \
      widget$ID   subcommand  -key value   makeCallback
```

Figure 15.1: How the `tcl` function maps its arguments

its ID component), converts R `key=value` pairs into `-key value` options for Tcl, and adjusts any callback functions. The `tcl` function uses position to create its command, the order of the subcommands needs to match that of the Tk API.

Often, the R object is first, but this is not always the case. As named arguments are only for the `-key value` expansion, we follow the Tcl language and call the arguments “options” in the following. The `tcl` function returns an object of class `tclObj`.

15.2 Constructors

In this Chapter, we will stick to a few basic widgets: labels and buttons; and top-level containers to illustrate the basic usage of `tcltk`, leaving for later more detail on containers and widgets.

Unlike GTK+, say, the construction of widgets in `tcltk` is tightly linked to the widget heirarchy. Tk widgets are constructed as children of a parent container with the parent specified to the constructor. When the Tk shell, `wish`, is used or the Tk package is loaded through the Tcl command `package require Tk`, a top level window named “.” is created. In the variable name `.label`, from above, the dot refers to the top level window. In `tcltk` a top-level window is created separately through the `tktoplevel` constructor, as with

```
w <- tktoplevel()
```

Top-level windows will be explained in more detail in Section 16. For now, we just use one to construct a label widget. Like all constructors but a toplevel window one, the label constructor (`ttklabel`) requires a specification of the parent container (`w`) and any other options that are desired. A typical usage would look like:

```
l <- ttklabel(w, text="label text")
```

Options The first argument of a constructor is the parent container, subsequent arguments are used to specify the options for the constructor given as `key=value` pairs. The Tk API lists these options along with their description.

For a simple label, the following options are possible: `anchor`, `background`, `font`, `foreground`, `justify`, `padding`, `relief`, `text`, and `wrlength`. This is

in addition to the standard options `class`, `compound`, `cursor`, `image`, `state`, `style`, `takefocus`, `text`, `textvariable`, `underline`, and `width`. (Although clearly lengthy, this list is significantly reduced from the options for `tklabel` where options for the many style properties are also included.)

Many of the options are clear from their name. The `padding` argument allows the specification of space in pixels between the text of the label and the widget boundary. This may be set as four values `c(left, top, right, bottom)`, or fewer, with `bottom` defaulting to `top`, `right` to `left` and `top` to `left`. The `relief` argument specifies how a 3-d effect around the label should look if specified. Possible values are `"flat"`, `"groove"`, `"raised"`, `"ridge"`, `"solid"`, or `"sunken"`.

The functions `tkcget`, `tkconfigure` Option values may be set through the constructor, or adjusted afterwards by `tkconfigure`. A listing (in Tcl code) of possible options that can be adjusted may be seen by calling `tkconfigure` with just the widget as an argument.

```
head(as.character(tkconfigure(l)))      # first 6 only
```

```
[1] "-background frameColor FrameColor {} {}"
[2] "-foreground textColor TextColor {} {}"
[3] "-font font Font {} {}"
[4] "-borderwidth borderWidth BorderWidth {} {}"
[5] "-relief relief Relief {} {}"
[6] "-anchor anchor Anchor {} {}"
```

The `tkcget` function returns the value of an option (again as a `tclobj` object). The option can be specified two different ways. Either using the Tk style of a leading dash or using the convention that `NULL` values mean to return the value, and not set it.

```
tkcget(l, "-text")                      # retrieve text property
```

```
<Tcl> label text
```

```
tkcget(l, text=NULL)                    # alternate syntax
```

```
<Tcl> label text
```

Coercion to character The `tclobj` objects can be coerced to character class two ways. The conversion through `as.character` breaks the return value along whitespace:

```
as.character(tkcget(l, text=NULL))
```

```
[1] "label" "text"
```

15. TCL Tk: OVERVIEW

The `tclvalue` function can also be used to extract the value from a `tclObj`, in this case not breaking along white space.

```
tclvalue(tkcget(1, text=NULL))
```

```
[1] "label text"
```

Buttons Buttons are constructed using the `ttkbutton` constructor.

```
b <- ttkbutton(w, text="click me")
```

Buttons and labels share many of the same options. However, buttons have a `command` option to specify a callback for when it is clicked. Callbacks will be explained in Section 15.3. Furthermore, buttons have the option `default` to specify which button of a dialog, by default, will get the Return signal when the enter key is pressed. A callback can then be set to respond to this signal. This value for `default` may be "active", indicating the button will get the signal; "normal"; or "disabled", to draw the button without space to indicate it

tkwidget Constructors call the `tkwidget` function which returns an object of class `tkwin`. (In Tk the term "window" is used to refer to the drawn widget and not just a top-level window)

```
str(b)
```

```
List of 2
 $ ID : chr ".2.2"
 $ env:<environment: 0x2ab5f88>
- attr(*, "class")= chr "tkwin"
```

The returned widget objects are lists with two components an ID and an environment. The ID component keeps a unique ID of the constructed widget. This is a character string, such as ".1.2.1" coming from the widget heirarchy of the object. This value is generated behind the scenes by the `tcltk` package using numeric values to keep track of the heirarchy. The `env` component contains an environment that keeps track of subwindows, the parent window and any callback functions. This helps ensure that any copies of the widget refer to the same object (Dalgaard). As the construction of a new widget requires the ID and environment of its parent, the first argument to `tkwidget`, `parent`, must be an R Tk object, not simply its character ID, as is possible for the `tcl` function. The latter is useful in a callback, as only the ID may be known to the callback function.

Geometry managers

As with Qt, when a new widget is constructed it is not automatically mapped. Tk uses geometry managers to specify how the widget will be drawn within

the parent container. We will discuss two such geometry managers in Section 16, but for now, we note that the simplest way to view a widget in its parent window is through `tkpack`:

```
tkpack(l)
tkpack(b)
```

This command packs the widgets into the top-level window (the parent in this case) in a box-like manner. Unlike GTK+ more than one child can be packed into a top-level window, although we don't demonstrate this further, as later we will use an intermediate `ttkframe` box container so that themes are properly displayed.

Tcl variables

For several Tk widgets, there is an option `textvariable` for a Tcl variable. These variables are dynamically bound to the widget, so that changes to the variable are propagated to the GUI. (The Tcl variable is a model and the widget a view of the model.) The basic functions involved are `tclVar` to create a Tcl variable, `tclvalue` to get the assigned value and `tclvalue<-` to modify the value.

```
textvar <- tclVar("another label")
l2 <- ttklabel(w, textvariable=textvar)
tkpack(l2)
tclvalue(textvar)
```

```
[1] "another label"
```

```
tclvalue(textvar) <- "new text"
```

The advantages of Tcl variables are like those of the MVC paradigm – a single data source can have its changes propagated to several widgets automatically. If the same text is to appear in different places, their usage is recommended. One disadvantage, is that in a callback, the variable is not passed to the callback and must be found through R's scoping rules.

Colors and fonts

The label color can be set through its `foreground` property. Colors can be specified by name – for common colors – or by hex RGB values which are common in web programming.

```
tkconfigure(l, foreground="red")
tkconfigure(l, foreground="#00aa00")
```

To find the hex RGB value, one can use the `rgb` function to create RGB values from intensities in $[0,1]$. The R function `col2rgb` can translate a named color into RGB values. The `as.hexmode` function will display an integer in hexadecimal notation.

Table 15.1: Standard font names defined by a theme.

Standard font name	Description
TkDefaultFont	Default font for all GUI items not otherwise specified
TkTextFont	Font for text widgets
TkFixedFont	Fixed-width font
TkMenuFont	Menu bar fonts
TkHeadingFont	Font for column headings
TkCaptionFont	Caption font (dialogs)
TkSmallCaptionFont	Smaller caption font
TkIconFont	Icon and text font

Fonts Fonts are more involved than colors. Tk version 8.5 made it more difficult to change style properties of individual widgets. This following the practice of centralizing style options for consistency, ease of maintaining code and ease of theming. To set a font for a label, rather than specify the font properties, one configures the font attribute using a pre-defined font name, such as

```
tkconfigure(l, font="TkFixedFont")
```

The "TkFixedFont" value is one of the standard font names, in this case to use a fixed-width font. A complete list of the standard names is provided in Table 15.2. Each theme sets these defaults accordingly. The `tkfont.create` function can be used to create a new font, as with the following commands:

```
tkfont.create("ourFont", family="Helvetica", size=12,
              weight="bold")
```

```
<Tcl> ourFont
```

```
tkconfigure(l, font="ourFont")
```

Available font families are system dependent. Only "Courier", "Times" and "Helvetica" are guaranteed to be there. A list of available font families is returned by the function `tkfont.families`. Figure 15.2 shows a display of some available font families on a Mac OS X machine. See Example 17.7 for details.

The arguments for `tkfont.create` are optional. The `size` argument specifies the pixel size. The `weight` argument can be used to specify "bold" or "normal". Additionally, a `slant` argument can be used to specify either "roman" (normal) or "italic". Finally, `underline` and `overstrike` can be set with a TRUE or FALSE value.

Font metrics The average character size is important in setting the width and height of some components. The can be found through the `tkfont.measure` and `tkfont.metrics` functions as follows:

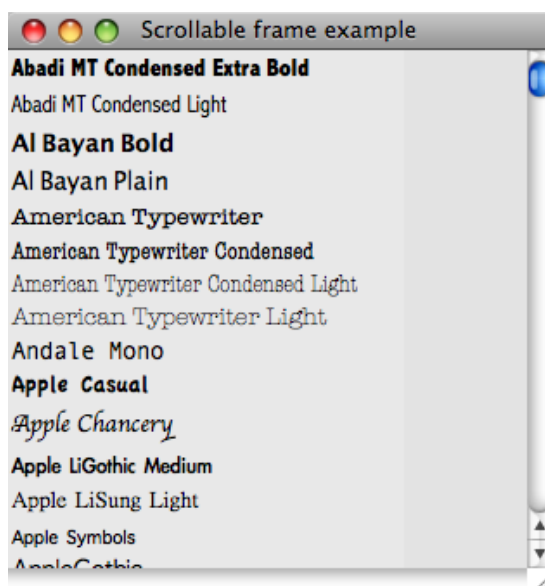


Figure 15.2: A scrollable frame widget (cf. Example 17.7) showing the available fonts on a system.

```
tmp <- tkfont.measure("TkTextFont",paste(c(0:9,LETTERS),collapse=""))
fontWidth <- ceiling(as.numeric(tclvalue(tmp))/36)
tmp <- tkfont.metrics("TkTextFont","linespace"=NULL)
fontHeight <- as.numeric(tclvalue(tmp))
c(width=fontWidth, height=fontHeight)
```

```
width height
    9      16
```

Images

Many tcltk widgets, including both labels and buttons, can show images. In these cases, either with or without an accompanying text label. Constructing images to display is similar to constructing new fonts, in that a new image object is created and can be reused by various widgets. The `tkimage.create` function is used to create image objects. The following command shows how an image object can be made from the file `tclp.gif` in the current directory:

```
tkimage.create("photo", "::img::tclLogo", file = "tclp.gif")
```

```
<Tcl> ::img::tclLogo
```

The first argument, "photo" specifies that a full color image is being used. This option could also be "bitmap" but that is more a legacy option. The second argument specifies the name of the object. We follow the advice of the Tk manual and preface the name with `::img::` so that we don't inadvertently overwrite any existing Tcl commands. The third argument `file` specifies the graphic file. The basic Tk image command only can show only GIF and PPM/PNM images. Unfortunately, not many R devices output in these formats. (The GDD device driver can.) One may need system utilities to convert to the allowable formats or install add-on Tcl packages.

To use the image, one can specify the name for the image option.

```
l <- ttklabel(w, image="::img::tclLogo", text="logo text",
             compound = "top")
```

By default the text will not show. The `compound` argument takes a value of either "text", "image" (default), "center", "top", "left", "bottom", or "right" specifying where the label is in relation to the text.

Image manipulation Once an image is created, there are several options to manipulate the image. These are found in the Tk man page for photo, not image. For instance, to change the palette so that instead of fullcolor only 16 shades of gray are used to display the icon, one could issue the command

```
tkconfigure("::img::tclLogo", palette=16)
```

Another useful manipulation to draw attention to an image is to change the `gamma` value when something happens, such as a mouse-over event (cf. Example 16.4).

Themes

The themed widgets have a style that determines how they are drawn. The separation of style properties from the widget, as opposed to having these set for each construction of a widget, makes it much easier to change the look of a GUI and easier to maintain the code. A collection of styles makes up a theme. The available themes depend on the system. The default theme should enable a GUI to have the native look and feel of the operating system. (This was definitely not the case for the older Tk widgets.) There is no built in command to return the theme, so we use `.Tcl` to call the appropriate Tcl command. The `names` sub command will return the available themes and the `use` sub command can be used to set the theme.

```
.Tcl("ttk::style theme names")
```

```
<Tcl> aqua clam alt default classic
```

```
.Tcl("ttk::style theme use classic")
```

The writing of themes will not be covered, but in Example 16.4 we show how to create a new style for a button.

The example we have shown so far, would not look quite right, as the toplevel window is not a themed widget. To work around that, a `ttkframe` widget is usually used to hold the child components of the top-level window. The following shows how to place a frame inside the window, with some arguments to be explained later that allow it to act reasonably if the window is resized.

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12)) # Some breathing room
tkpack(f, expand=TRUE, fill="both")    # For window expansion
l <- ttklabel(f, image="::img::tclLogo", text="label",
             compound="top")
tkpack(l)
```

Window properties and state: `tkwininfo`

Widgets have options which can be set through `tkconfigure` and additionally, when mapped, the “window” they are rendered to has properties, such as a class or size. These properties are queried through the `tkwininfo` function. There are several such properties, and may take different forms. If the API is of the form

```
wininfo subcommand_name window
```

the specification to `tkwininfo` is in the same order (the widget is not the first argument). For instance, the class of a label is returned by the `class` subcommand as

```
tkwininfo("class", l)
```

```
<Tcl> TLabel
```

The window, in this example, `l`, can be specified as an R object, or by its character ID. This is useful, as the return value of some functions is the ID. For instance, the `children` subcommand returns IDs. Below the `as.character` function will coerce these into a vector of IDs.

```
(children <- tkwininfo("children",f))
```

```
<Tcl> .3.1.1
```

```
sapply(as.character(children), function(i) tkwininfo("class", i))
```

```
$ '.3.1.1'
```

```
<Tcl> TLabel
```

There are several possible subcommands, here we list a few. The *tkwininfo* geometry sub command returns the location and size of the widgets' window in the form `width x height + x + y`; the sub commands *tkwininfo* height, *tkwininfo* width, *tkwininfo* x, or *tkwininfo* y can be used to return just those parts. The *tkwininfo* exists command returns 1 (TRUE) if the window exists and 0 otherwise; the *tkwininfo* ismapped sub command returns 1 or 0 if the window is currently mapped (drawn); the *tkwininfo* viewable sub command is similar, only it checks that all parent windows are also mapped. For traversing the widget heirarchy, one has available the *tkwininfo* parent sub command which returns the immediate parent of the component, *tkwininfo* toplevel which returns the ID of the top-level window, and *tkwininfo* children which returns the IDs of all the immediate child components, if the object is a container, such as a top-level window.

15.3 Events and Callbacks

The button widget has the `command` option for assigning a callback for when the user clicks the mouse button on the button. In addition to this, one can specify callbacks for many other events that the user may initiate.

Callbacks

The `tcltk` package implements callbacks in a manner different from Tk, as the callback functions are R functions, not Tk procedures. This is much more convenient, but introduces some slight differences. In `tcltk` these callbacks can be expressions (unevaluated calls) or functions. We use only the latter, for more clarity. The basic callback function need not have any arguments. For instance, here we show how to print a message when the user clicks a button:

```
w <- tkoplevel()
callback <- function() print("hi")
b <- ttkbutton(w, text="Click me", command = callback)
tkpack(b)
```

The callback's return value is generally not important, although we shall see with the validation framework, discussed in Section 17.2, it can matter.² As well, in Tk callbacks are evaluated in the global environment, but this is not so in `tcltk`, which respects the callback's scope.

²The difference in processing of return values can make porting some Tk code to `tcltk` difficult

Events

When a user interacts with a GUI, they initiate events. The `tcltk` package allows the programmer to bind callbacks to these events, through the `tkbind` function. This function is called as `tkbind(tag, events, command)`. The `command` is a callback, as described above.

The `tag` argument allows for quite a bit of flexibility. It can be:

- the name of a widget** , in which case the command will be bound to that widget;
- a top-level window** , in which case the command will be bound to the event for the window and all its internal widgets;
- a class of widget** , such as `"TButton"`, in which case all such widgets will get the binding; or
- the value "all"** , in which case all widgets in the application will get the binding.

The possible events (or sequences of events) vary from widget to widget. Events can be specified in a few ways. A single keypress event, can be assigned by specifying the ASCII character generated. For instance, to bind to `C` for the "Click me" button above using the same callback could be done with

```
tkbind(b, "C", callback)
tkfocus(b)
```

The `tkfocus` function is used to set the focus to the button so that it will receive the keypress.

Events with modifiers More complicated events can be described with the pattern

`<modifier-modifier-type-detail>`.

Examples of a type are `<KeyPress>`, `<ButtonPress>`. The event `<Control-c>` has the type `c` and modifier `Control`. Whereas `<Double-Button-1>` also has the detail `1`. The full list of modifiers and types are described in the man page for `bind`. Some familiar modifiers are `Control`, `Alt`, `Button1` (or its shortening `B1`), `Double` and `Triple`. The event types are the standard X event types along with some abbreviations. These are also listed in the `bind` man page. Some commonly used ones are `ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `FocusIn`, and `FocusOut`.

Window manager events Some events are based on window manager events. The `<Configure>` event happens when a component is resized. The `<Map>` and `<Unmap>` events happen when a component is drawn or undrawn.

Virtual events Finally, the event may be a “virtual event.” These are represented with `<<EventName>>`. There are predefined virtual events listed in the event man page. These include `<<MenuSelect>>` when working with menus, `<<Modified>>` for text widgets, `<<Selection>>` for text widgets, and `<<Cut>>`, `<<Copy>>` and `<<Paste>>` for working with the clipboard. New virtual events can be produced with the `tkevent.add` function. This takes at least two arguments, an event name and a sequence that will initiate that event. The event man page has these examples coming from the Emacs world:

```
tkevent.add("<<Paste>>", "<Control-y>")
tkevent.add("<<Save>>", "<Control-x><Control-s>")
```

In addition to virtual events occurring when the sequence is performed, the `tkevent.generate` can be used to force an event for a widget. This function requires a widget (or its ID) and the event name. Other options can be used to specify substitution values, described below. To illustrate, this command will generate the `<<Save>>` event for the button `b`:

```
tkevent.generate(b, "<<Save>>")
```

In `tcltk` only one callback can be associated with a widget and event through the call `tkbind(widget,event,callback)`. (Although, callbacks for the widget associated with classes or toplevel windows can differ.) Calling `tkbind` another time will replace the callback. To remove a callback, simply assign a new callback which does nothing.³

% Substitutions

Tk provides a mechanism called *percent substitution* to pass information about the event to callbacks bound to the event. The basic idea is that in the Tcl callback expressions of the type `%X`, for different characters `X`, will be replaced by values coming from the event. In `tcltk`, if the callback function has an argument `X`, then that variable will correspond to the value specified by `%X`. The complete list of substitutions is in the `bind` man page. Useful ones are `x` and `X` to specify the relative or absolute *x*-position of a mouse click (the difference can be found through the `rootx` property of a widget), `y` and `Y` for the *y*-position, `k` and `K` for the keycode (ASCII) and key symbol of a `<KeyPress>` event, and `W` to refer to the ID of the widget that signaled the event the callback is bound to. Example 15.1 will illustrate some of these.

The after command The Tcl command `after` will execute a command after a certain delay (specified in milliseconds as an integer) while not interrupting

³This event handling can prevent being able to port some Tk code into `tcltk`. In those cases, one may consider sourcing in Tcl code directly.

the control flow while it waits for its delay. The function is called in a manner like:

```
ID <- tcl("after", 1000, function() print("1 second passed"))
```

The ID returned by after may be used to cancel the command before it executes. To execute a command repeatedly, can be done along the lines of:

```
afterID <- ""; someFlag <- TRUE
repeatCall <- function(ms=10000, f, w) {
  afterID <- tcl("after", ms, function() {
    if(someFlag) {
      f()
      afterID <- repeatCall(ms, f, w)
    } else {
      tcl("after", "cancel", afterID)
    }
  })
}
repeatCall(100, function() print("running"), w)
```

The flag allows for the cancellation of the repeated call.

Example 15.1: Drag and Drop

This long example shows how to implement drag and drop between two widgets. Steps are needed to make a widget a drop source, and other steps are needed to make a widget a drop target. The basic idea is that when a value is being dragged, virtual events are generated for the widget the cursor is over. If that widget has callbacks bound to these events, then the drag and drop can be processed. The idea for the code below originated with <http://wiki.tcl.tk/416>.

To begin, we create a simple GUI to hold three widgets. We use buttons for drag and drop, but only because we haven't yet discussed more natural widgets such as the text widgets.

```
w <- tkoplevel()
bDrag <- ttkbutton(w, text="Drag me")
bDrop <- ttkbutton(w, text="Drop here")
tkpack(bDrag)
tkpack(ttklabel(w, text="Drag over me"))
tkpack(bDrop)
```

Before beginning, we define three global variables that can be shared among drop sources to keep track of the drag and drop state. A more elegant example might store these in an environment.

```
.dragging <- FALSE           # currently dragging?
.lastWidgetID <- ""         # last widget dragged over
.dragValue <- ""            # value to transfer
```

To set up a drag source, we bind to three events: a mouse button press, mouse motion, and a button release. For the button press, we set the values of the three global variables.

```
tkbind(bDrag, "<ButtonPress-1>", function(W) {  
    .dragging <- TRUE  
    .lastWidgetID <- as.character(W)  
    .dragValue <- as.character(tkcget(W, text=NULL))  
})
```

For mouse motion, we do several things. First we set the cursor to indicate a drag operation. The choice of cursor is a bit outdated. The commented code shows how one can put in a custom cursor from an xbm file, but this doesn't work for all platforms (e.g., OS X). After setting the cursor, we find the ID of the widget the cursor is over. This uses `tkwinfo` to find the widget containing the x,y -coordinates of the cursor position. We then generate the `<<DragOver>>` virtual event for this widget, and if this widget is different from the previous last widget, we generate the `<<DragLeave>>` virtual event.

```
tkbind(bDrag, "<B1-Motion>", function(W, X, Y) {  
    if(!.dragging) return()  
    ## see cursor help page in API for more options  
    ## For custom cursors cf. http://wiki.tcl.tk/8674.  
    tkconfigure(W, cursor="coffee_mug") # set cursor  
  
    w = tkwinfo("containing", X, Y) # widget mouse is over  
    if(as.logical(tkwinfo("exists", w))) # does widget exist?  
        tkevent.generate(w, "<<DragOver>>")  
  
    ## generate drag leave if we left last widget  
    if(as.logical(tkwinfo("exists", w)) &&  
        length(as.character(w)) > 0 &&  
        length(as.character(.lastWidgetID)) > 0  
    ) {  
        if(as.character(w)[1] != .lastWidgetID)  
            tkevent.generate(.lastWidgetID, "<<DragLeave>>")  
    }  
    .lastWidgetID <- as.character(w)  
})
```

Finally, if the button is released, we generate the virtual events `<<DragLeave>>` and most importantly `<<DragDrop>>` for the widget we are over.

```
tkbind(bDrag, "<ButtonRelease-1>", function(W, X, Y) {  
    if(!.dragging) return()  
    w = tkwinfo("containing", X, Y)  
  
    if(as.logical(tkwinfo("exists", w))) {
```

```

    tkevent.generate(w, "<<DragLeave>>")
    tkevent.generate(w, "<<DragDrop>>")
    tkconfigure(w, cursor="")
  }
  .dragging <- FALSE
  tkconfigure(W, cursor="")
})

```

To set up a drop target, we bind callbacks for the virtual events generated above to the widget. For the `<<DragOver>>` event we make the widget active so that it appears ready to receive a drag value.

```

tkbind(bDrop, "<<DragOver>>", function(W) {
  if(.dragging)
    tkconfigure(W, default="active")
})

```

If the drag event leaves the widget without dropping, we change the state back to normal.

```

tkbind(bDrop, "<<DragLeave>>", function(W) {
  if(.dragging) {
    tkconfigure(W, cursor="")
    tkconfigure(W, default="normal")
  }
})

```

Finally, if the `<<DragDrop>>` virtual event occurs, we set the widget value to that stored in the global variable `.dragValue`.

```

tkbind(bDrop, "<<DragDrop>>", function(W) {
  tkconfigure(W, text=.dragValue)
  .dragValue <- ""
})

```


Tcl Tk: Containers and Layout

16.1 Top-level windows

Top level windows are created through the `tktoplevel` constructor. The arguments `width` and `height` may be specified to give a requested size. Negative values means the window will not request any size. Top-level windows can have a menubar specified through the `menu` argument. Menus will be covered in Section 17.4.

The `tkdestroy` function can be called to destroy the window and its child components.

The Tk command `wm` is used to interact with top-level windows. This command has several subcommands, leading to `tcltk` functions with names such as `tkwm.title`, the function used to set the window title. As with all such functions, either the top-level window object, or its ID must be the first argument. In this case, the new title is the second.

When a top-level window is constructed there is no option for it not to be shown. However, one can use the `tclServiceMode` function to suspend/resume drawing of any widget through Tk. This function takes a logical value indicating the updating of widgets should be suspended. One can set the value to `FALSE`, initiate the widgets, then set to `TRUE` to display the widgets. After a window is drawn. To iconify an already drawn window can be done through the `tkwm.withdraw` function and reversed with the `tkwm.deiconify` function. Together these can be useful to use in the construction of complicated GUIs, as the drawing of the widgets can seem slow. (The same can be done through the `tkwm.state` function with an option of `"withdraw"` or `"normal"`.)

Window sizing The preferred size of a top-level window can be configured through the `width` and `height` options. The absolute size and position of a top-level window in pixels can be queried or specified through the `tkwm.geometry` function. The geometry is specified as a string in the form `=w x h + x + y` (or `-`) where any of `=`, `wxh` or `+x+y` can be omitted. The value

for *x* (if using *+*) indicates how many pixels to the right from the left edge should the window be placed (if using *-* then the left side of the screen is used as a reference). For *y* the top (or bottom) of the screen is the reference.

The `ttksizegrip` widget can be used to add a visual area (usually the lower right corner) for the user to grab on to with their mouse for resizing the window. On some OSes (e.g., Mac OS X) these are added by the window manager automatically.

The `tkwm.resizable` function can be used to prohibit the resizing of a top-level window. The syntax allows either the width or height to be constrained. The following command would prevent resizing of both the width and height of the toplevel window *w*.

```
tkwm.resizable(w, FALSE, FALSE)    # width first
```

When a window is resized, you can constrain the minimum and maximum sizes with `tkwm.minsize` and `tkwm.maxsize`. The aspect ratio (width/height) can be set through `tkwm.aspect`.

For some uses it may be desirable to not have the window manager decorate the window with a title bar etc. Tooltips, for example, can be constructed using this approach. The command `tkoplevel wm overrideredirect logical` takes a logical value indicating if the window should be decorated. Though, not all window managers respect this.

bindings Bindings for top-level windows are propagated down to all of their child widgets. If a common binding is desired for all the children, then it need only be specified once for the top-level window.

The `tkwm.protocol` function (not `tkbind`) is used to assign commands to window manager events, most commonly, the delete event when the user clicks the close button on the windows decorations. A top-level window can be removed through the `tkdestroy` function, or through the user clicking on the correct window decorations. When the window decoration is clicked, the window manager issues a "WM_DELETE_WINDOW" event. To bind to this, a command of this form `tkwm.protocol(win,"WM_DELETE_WINDOW", callback)` is used.

To illustrate, if *w* is a top-level window, and *e* a text entry widget (cf. Section 17.2) then the following snippet of code would check to see if the text widget has been modified before closing the window. This uses a modal message box described in Section 17.6.

```
tkwm.protocol(w,"WM_DELETE_WINDOW", function() {
    modified <- tcl(e, "edit", "modified")
    if(as.logical(modified)) {
        response <-
            tkmessageBox(icon="question",
                        message="Really close?",
                        detail="Changes need to be saved",
```



```

        type="yesno",
        parent=w)
    if(as.character(response) == "no")
        return()
    }
    tkdestroy(w)                                # otherwise close
})

```

Sometimes, say with dialogs, a top-level window should be related to a different top-level window. The function `tkwm.transient` allows one to specify the master window as its second argument. The new window will mirror the state of the master window, including if the master is withdrawn.

A window can be made to always be the topmost window through the `attributes` subcommand of the `wm` command. However, there is no direct `tcltk` function, so if `w` was to be on top, one would use the `tcl` function as follows:

```
tcl("wm", "attributes", w, topmost=TRUE)
```

16.2 Frames

The `ttkframe` constructor produces a themable container that can be used to organize visible components within a GUI. It is often the first thing packed within a top-level window. (As in the example of Section 15.2.)

The options include `width` and `height` to set the requested size, `borderwidth` to specify a border around the frame of a given width, and `relief` to set the border style. The value of `relief` is chosen from the default "flat", "groove", "raised", "ridge", "solid", or "sunken". The `padding` option can be used to put space within the border between the border and subsequent children.

Label Frames

The `ttklabelframe` constructor produces a frame with an optional label that can be used to set off and organize components of a GUI. The label is set through the option `text`. Its position is determined by the option `labelanchor` taking values labeled by compass headings (combinations of `n`, `e`, `w`, `s`). The default is theme dependent, although typically "nw" (upper left).

Separators To use a single line to separate out areas in a GUI, the `ttkseparator` widget can be used. The lone widget-specific option is `orient` which takes values of "horizontal" (the default) or "vertical". This widget must be told to stretch when added to a container, as described in the next section.

16.3 Geometry Managers

Tcl uses *geometry managers* to place child components within their parent windows. There are three such managers, but we describe only two, leaving the lower-level `place` command for the official documentation. The use of geometry managers, allows Tk to quickly reallocate space to a GUI's components when it is resized. The `tkpack` function will place children into their parent in a box-like manner. We have seen in several examples throughout the text, that through the use of nested boxes, one can construct quite flexible layouts, and Example 16.2 will illustrate that once again. When simultaneous horizontal and vertical alignment of child components is desired, the `tkgrid` function can be used to manage the components.

A GUI may use a mix of `pack` and `grid` to manage the child components, but all siblings in the widget hierarchy must be managed the same way. Mixing the two will typically result in a lockup of the R session.

Pack

We have illustrated how `tkpack` can be used to manage how child components are viewed within their parent. The basic usage `tkpack(child)` will pack in the child components from top to bottom. The `side` option can take a value of "left", "right", "top" (default), or "bottom" to adjust where the children are placed. These can be mixed and matched, but sticking to just one direction is typical, with nested frames to give additional flexibility.

after, before The `after` and `before` options can be used to place the child before or after another component. These are used as with `tkpack(child1, after=child2)`. The object `child2` can be an R object or its ID. The latter might be useful, say when all the children are listed using the command `tkwininfo("children",parent)` which returns the IDs of the immediate child components.

padding In addition to the `padding` option for a frame container, the `ipadx`, `ipady`, `padx`, and `pady` options can be used to add space around the child components. Figure 16.1 has an example. The `x` and `y` indicate left-right space or top-bottom space. The `i` stands for internal padding that is left on the sides or top and bottom of the child within the border, for `padx` the external padding added around the border of the child component. The value can be a single number or pair of numbers for asymmetric padding.

This sample code shows how one can easily add padding around all the children of the frame `f` using the `tkpack "configure"` subcommand.

Cavity model The packing algorithm, as described in the Tk documentation, is based on arranging where to place a slave into the rectangular unal-

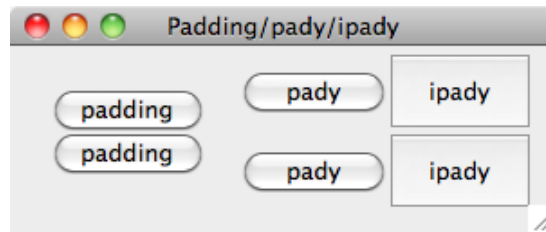


Figure 16.1: Various ways to put padding between widgets using a box container and `tkpack`. The `padding` option for the box container puts padding around the cavity for all the widgets. The `pady` option for `tkpack` puts padding around the top and bottom on the border of each widget. The `ipady` option for `tkpack` puts padding within the top and bottom of the border for each child (modifying the theme under Mac OS X).

located space called a cavity. We use the nicer terms child component and box to describe these. When a child is placed inside the box, say on the top, the space allocated to the child is the rectangular space with width given by the width of the box, and height the sum of the requested height of the child plus twice the `ipady` amount (or the sum if specified with two numbers). The packer then chooses the dimension of the child component, again from the requested size plus the `ipad` values for `x` and `y`. These two spaces may, of course, have different dimensions.

anchor By default, the child will be placed centered along the edge of the box within the allocated space and blank space, if any, on both sides. If there is not enough space for the child in the allocated space, the component can be drawn oddly. Enlarging the top-level window can adjust this. When there is more space in the box than requested by the child component, there are other options. The `anchor` option can be used to anchor the child to a place in the box by specifying one of the valid compass points (eg. `"n"` or `"se"`) leaving blank space around the child. External padding between the child and the box can be set through the `padx` and `pady` options.

expand, fill When there is more space in the original box than needed by the children the extra space will be left blank unless some children have the option `expand` set to `TRUE`. In this case, the extra space is allocated evenly to each child with this set. The `fill` option is often used when `expand` is set. The `fill` option is used to base the size of the child on the available cavity in the box – not on the requested size of the child. The `fill` option can be `"x"`, `"y"` or `"both"`. The first two expanding the child's size in just one direction, the latter in both.

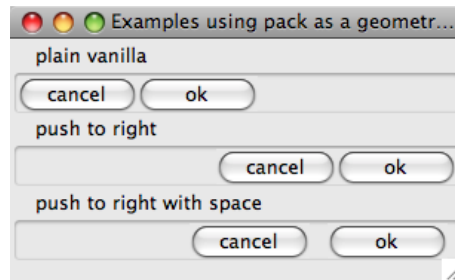


Figure 16.2: Demonstration of using `tkpack` options showing effects of using the `side` and `padx` options to create dialog buttons.

forget Child components can be forgotten by the window manager, unmaping them but not destroying them, with the `tkpack forget` subcommand, or the convenience function `tkpack.forget`. After a child component is removed this way, it can be re-placed in the GUI using a geometry manager. In `gWidgetstcltk` this is used to create a `gexpandgroup` container, as such a container is not provided by Tk.

Introspection The subcommand `tkpack slaves` will return a list of the child components packed into a frame. Coercing these return values to character via `as.character` will produce the IDs of the child components. The subcommand `tkpack info` will provide the packing info for a child.

Example 16.1: Packing dialog buttons

This example shows how one can pack in action buttons, such as when a dialog is created.

The first example just uses `tkpack` without any arguments except the `side` to indicate the buttons are packed in left to right, not top to bottom.

```
f1 <- ttklabelframe(f, text="plain vanilla")
tkpack(f1, expand=TRUE, fill="x")
l <- function(f)
  list(ttkbutton(f, text="cancel"), ttkbutton(f, text="ok"))
QT <- sapply(l(f1), function(i) tkpack(i, side="left"))
```

Typically the buttons are right justified. One way to do this is to pack in using `side` with a value of `"right"`. This shows how to use a blank expanding label to take up the space on the left.

```
f2 <- ttklabelframe(f, text="push to right")
tkpack(f2, expand=TRUE, fill="x")
tkpack(ttklabel(f2, text=" "), expand=TRUE, fill="x", side="left")
QT <- sapply(l(f2), function(i) tkpack(i, side="left"))
```



Figure 16.3: Example of a simple dialog

Finally, we add in some padding to conform to Apple's design specification that such buttons should have a 12 pixel separation.

```
f3 <- ttklabelframe(f, text="push to right with space")
tkpack(f3, expand=TRUE, fill="x")
tkpack(ttklabel(f3, text=" "), expand=TRUE, fill="x", side="left")
QT <- sapply(1(f3), function(i) tkpack(i, side="left", padx=6))
```

Example 16.2: A non-modal dialog

This example shows how to use a window, frames, labels, buttons, icons, packing and bindings to create a non-modal dialog.

Although not written as a function, we set aside the values that would be passed in were it.

```
title <- "message dialog"
message <- "Do you like tcltk so far?"
parent <- NULL
QT <- tkimage.create("photo", "::img::tclLogo",
                     file = system.file("images", "tclp.gif",
                                         package="ProgGUIinR"))
```

The main top-level window is then given a title, then withdrawn while the GUI is created.

```
w <- tktoplevel(); tkwm.title(w, title)
tkwm.state(w, "withdrawn")
f <- ttkframe(w, padding=c(3, 3, 12, 12))
tkpack(f, expand=TRUE, fill="both")
```

As usual, we added a frame so that any themes are respected.

If the parent is non-null and is viewable, then the dialog is made transient to a parent. The parent need not be a top-level window, so `tkwinfo` is used to find the parent's top-level window. For Mac OS X, we use the `notify` attribute to bounce the dock icon until the mouse enters the window area.

```
if(!is.null(parent)) {
  parentWin <- tkwinfo("toplevel", parent)
```

```
if(as.logical(tkwininfo("viewable", parentWin))) {  
    tkwm.transient(w, parent)  
    ## effects OS X only now  
    tcl("wm","attributes",parentWin, notify=TRUE) # bounce  
    tkbind(parentWin,"<Enter>", function()  
        tcl("wm","attributes",parentWin, notify=FALSE)) #stop  
    }  
}
```

We will use a standard layout for our dialog with an icon on the left, a message and buttons on the right. We pack the icon into the left side of the frame,

```
l <- ttklabel(f, image="::img::tclLogo", padding=5) # recycle  
tkpack(l,side="left")
```

A nested frame will be used to layout the message area and button area. Since the tkpack default is to pack in top to bottom, no side specification is made.

```
f1 <- ttkframe(f)  
tkpack(f1, expand=TRUE, fill="both")  
#  
m <- ttklabel(f1, text=message)  
tkpack(m, expand=TRUE, fill="both")
```

The buttons have their own frame, as they are layed out horizontally.

```
f2 <- ttkframe(f1)  
tkpack(f2)
```

The callback function for the OK button prints a message then destroys the window.

```
okCB <- function() {  
    print("That's great")  
    tkdestroy(w)  
}  
okButton <- ttkbutton(f2, text="OK", default="active")
```

We bind the callback to both a left mouse click on the button, and if the user presses return when the button has the focus. The default="active" argument, makes this button the one that gets the Return event when the return key is pressed.

```
tkbind(okButton, "<Button-1>", okCB)  
tkbind(okButton, "<Return>", okCB)  
cancelButton <- ttkbutton(f2, text="Cancel",  
    command=function() tkdestroy(w))  
tkpack(okButton, side="left", padx=12) # give some space  
tkpack(cancelButton)
```

Now we bring the dialog back from its withdrawn state, fix the size and set the focus on the OK button.

```
tkwm.state(w, "normal")
tkwm.resizable(w, FALSE, FALSE)
tkfocus(okButton)
```

Finally, the following bindings make the buttons look active when the keyboard focus is on them, generating a `FocusIn` event, then a `FocusOut` event. We make a binding for the top-level window, then within the callback check to see if the widget emitting the signal is of a themed button class.

```
isTButton <- function(W)
  as.character(tkwininfo("class",W)) == "TButton"
tkbind(w,"<FocusIn>", function(W) {
  if(isTButton(W)) tkconfigure(W,default="active")
})
tkbind(w,"<FocusOut>", function(W) {
  if(isTButton(W)) tkconfigure(W,default="normal")
})
```

Grid

The `tkgrid` geometry manager is used to place child widgets in rows and columns. In its simplest usage, a command like

```
tkgrid(child1, child2,..., childn)
```

will place the n children in a new row, in columns 1 through n . However, the specific row and column can be specified through the `row` and `column` options. Counting of rows and columns starts with 0. Spanning of multiple rows and columns can be specified with integers 2 or greater by the `rowspan` and `colspan` options. These options, and others can be adjusted through the `tkgrid.configure` function.

The `tkgrid.rowconfigure`, `tkgrid.columnconfigure` commands When the managed container is resized, the grid manager consults weights that are assigned to each row and column to see how to allocate the extra space. These weights are configured with the `tkgrid.rowconfigure` and `tkgrid.columnconfigure` functions through the option `weight`. The weight is a value between 0 and 1. If there are just two rows, and the first row has weight 1/2 and the second weight 1, then the extra space is allocated twice as much for the second row. The specific row or column must also be specified. Rows and columns are referenced starting with 0 not the usual R-like 1. So to specify a weight of 1 to the first row would be done with a command like:

```
tkgrid.rowconfigure(parent, 0, weight=1)
```

The sticky option When more space is available than requested by the child component, the `sticky` option can be used to place the widget into the grid. The value is a combination of the compass points "n", "e", "w", and "s". A specification "ns" will make the child component “stick” to the top and bottom of the cavity that is provided, similar to the `fill="y"` option for `tkpack`. A value of "news" will make the child component expand in all the direction like `fill="both"`.

Padding As with `tkpack`, `tkgrid` has options `ipadx`, `ipady`, `padx`, and `pady` to give internal and external space around a child.

Size The function `tkgrid.size` will return the number of columns and rows of the specified parent container that is managed by a grid. This can be useful when trying to position child components through the options `row` and `column`.

Forget To remove a child from the parent, the `tkgrid.forget` function can be used with the child object as its argument.

Example 16.3: Using `tkgrid` and `tkpack` to draw some world flags

This example shows how the `tkpack` and `tkgrid` geometry managers can be used to draw some of the world flags. For these, we consulted <https://www.cia.gov/library/publications/the-world-factbook/docs/flagsoftheworld.html>.

We will make the dimensions of the flags true to the flag proportions. These we found at <http://flagspot.net/flags/xf-size.html>. Here we define the proportions for the flags of interest.

```
dims <- cbind(Benin=2:3, Cameroon=2:3,Guinea=2:3, Mali=2:3,
              Bolivia=2:3, Lithuania=1:2,Congo=2:3, Guyana=1:2,
              Togo= 2:3)
```

This is a convenience function to create `tkframes` with different background colors. We use `tkframe` here – not `ttkframe` – as it has a `background` property.

```
makeColors <- function(parent)
  list(green = tkframe(parent, background="green"),
       red   = tkframe(parent, background="red"),
       yellow = tkframe(parent, background="yellow"))
```

This convenience function packs a frame into a top-level window.

```
makeTopLevel <- function(country) {
  w <- tktoplevel()
  tkwm.title(w, country)
  f <- ttkframe(w, padding=c(3,3,3,12))
```




Figure 16.4: Example of world flags to illustrate `tkpack` and `tkgrid` usage. The Mali flag uses `expand=TRUE` to allocate space evenly, `fill="both"` to have the child fill the space and `side="left"` to place the children, whereas Lithuania uses `side="top"`. The Benin flag takes advantage of `tkgrid` to layout the colors in a grid. The left color has `rowspan=2` set. The Togo flag could be done using just `grid`, but a mix is demonstrated.

```
tkpack(f, expand=TRUE, fill="both")
return(list(w=w, f= f, country=country))
}
```

Our first flags are Cameroon (GRY), Guinea (RYG), and Mali (GYR). These are flags with 3 equal vertical strips of color. We use `tkpack` with `side="left"` to pack in the colors from left to right. The `expand=TRUE` option causes extra space to be allocated equally to the three children, preserving the equal sizes in this case.

```
win <- makeToplevel("Cameroon")
w <- win$w; f <- win$f
l <- makeColors(f)
tkpack(l$green, l$red, l$yellow, expand=TRUE,
      fill="both", side="left")
```

To create Guinea's flag we simply move the green strip to the end.

```
## Guinea just moves colors around
tkpack("forget", l$green)
tkpack(l$green, expand=TRUE, fill="both", side="left")
tkwm.title(win$w, "Guinea")
```

For Mali, we flip the position of green and red. We pack them in relative to the yellow strip using the before and after options to tkpack.

```
tkpack("forget", l$green)
tkpack("forget", l$red)
tkpack(l$green, before=l$yellow, expand=TRUE, fill="both", side="left")
tkpack(l$red, after=l$yellow, expand=TRUE, fill="both", side="left")
tkwm.title(win$w, "Mali")
```

Lithuania is similar, only the stripes run horizontally. We pack from top to bottom to achieve this.

```
win <- makeTopLevel("Lithuania")
l <- makeColors(win$f)
tkpack(l$yellow, l$green, l$red, expand=TRUE, fill="both", side="top")
```

Benin's flag is better suited for the grid geometry manager. We use a combination of rowspan and colspan to get the proper arrangement. In this case, the proportions of the colors are achieved through equal weights when we configure the row and columns.

```
## benin is better suited for grid
win <- makeTopLevel("Benin")
l <- makeColors(win$f)
tkgrid(l$green, row=0, column=0, rowspan=2, sticky="news")
tkgrid(l$yellow, row=0, column=1, colspan=2, sticky="news")
tkgrid(l$red, row=1, column=1, colspan=2, sticky="news")
## use grid in equal sizes to get spacing right
tkgrid.rowconfigure(win$f, 0:1, weight=1)
tkgrid.columnconfigure(win$f, 0:2, weight=1)
```

Togo is trickier. We could use grid, as above, with the proper combinations of row and colspan. Instead we do this less directly to illustrate the mixing of the tkgrid and tkpack geometry managers.

```
win <- makeTopLevel("Togo")
f <- win$f
l <- makeColors(f)
upperR <- ttkframe(f); bottom <- ttkframe(f)
## upper left red rectangle
tkgrid(l$red, row=0, column=0, sticky="news")
tkgrid(upperR, row=0, column=1, sticky="news")
tkgrid(bottom, row=1, column=0, colspan=2, sticky="news")
## top right stripes
l1 <- makeColors(upperR)
tkpack(l1$yellow, expand=TRUE, fill="both", side="top")
tkpack(l1$green, expand=TRUE, fill="both", side="top")
## bottom stripes
l2 <- makeColors(bottom)
tkpack(l2$yellow, expand=TRUE, fill="both", side="top")
```

```
tkpack(l2$green, expand=TRUE, fill="both", side="top")
tkgrid.rowconfigure(f, 0:1, weight=1)
tkgrid.columnconfigure(f, 0, weight=8)
tkgrid.columnconfigure(f, 1, weight=10) # not quite uniform
```

Example 16.4: Using tkgrid to create a toolbar

Tk does not have a toolbar widget. Here we use tkgrid to show how we can add one to a top-level window in a manner that is not affected by resizing. We begin by packing a frame into a top-level window.

```
w <- tktoplevel(); tkwm.title(w, "Toolbar example")
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
```

Our example has two main containers: one to hold the toolbar buttons and one to hold the main content.

```
tbFrame <- ttkframe(f, padding=0)
contentFrame <- ttkframe(f, padding=4)
```

The tkgrid geometry manager is used to place the toolbar at the top, and the content frame below. The choice of sticky and the weights ensure that the toolbar does not resize if the window does.

```
tkgrid(tbFrame, row=0, column=0, sticky="we")
tkgrid(contentFrame, row=1, column=0, sticky="news")
tkgrid.rowconfigure(f, 0, weight=0)
tkgrid.rowconfigure(f, 1, weight=1)
tkgrid.columnconfigure(f, 0, weight=1)
## some example to pack into the content area
tkpack(ttklabel(contentFrame, text="Some content"))
```

Now to add some buttons to the toolbar. We first show how to create a new style for a button, slightly modifying the themed button to set the font and padding, and eliminate the border if the OS allows.

```
tcl("ttk::style","configure","Toolbar.TButton",
    font="helvetica 12", padding=0, borderwidth=0)
```

This makeIcon function finds stock icons from the gWidgets package and adds them to a button.

```
makeIcon <- function(parent, stockName, command=NULL) {
  iconFile <- system.file("images",
                          paste(stockName,"gif",sep="."),
                          package="gWidgets")
  if(nchar(iconFile) == 0) {
    b <- ttkbutton(parent, text=stockName, width=6)
  } else {
```

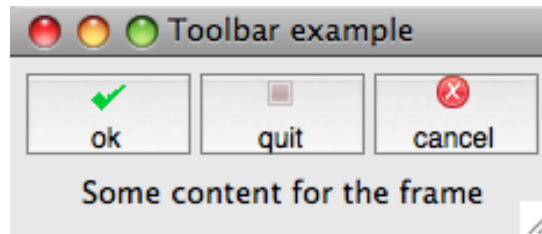


Figure 16.5: Illustration of using tkpack to make a toolbar.

```

iconName <- paste("::img::",stockName, sep="")
tkimage.create("photo", iconName, file = iconFile)
b <- ttkbutton(parent, image=iconName,
               style="Toolbar.TButton", text=stockName,
               compound="top", width=6)
if(!is.null(command))
    tkconfigure(b, command=command)
}
return(b)
}

```

To illustrate, we pack in some icons. Here we use tkpack. One does not use tkpack and tkgrid to manage children of the same parent, but these are children of tbFrame, not f.

```

tkpack(makeIcon(tbFrame, "ok"), side="left")
tkpack(makeIcon(tbFrame, "quit"), side="left")
tkpack(makeIcon(tbFrame, "cancel"), side="left")

```

These two bindings show how to slightly highlight the icon when the mouse is over that button, so that the user has some extra feedback.

```

changeGamma <- function(W, gamma=1.0) {
    if(as.character(tkwininfo("class",W)) == "TButton") {
        img <- tkcget(W,"image"=NULL)
        tkconfigure(img, gamma=gamma)
    }
}
tkbind(w,"<Enter>", function(W) changeGamma(W, gamma=0.5))
tkbind(w,"<Leave>", function(W) changeGamma(W, gamma=1.0))

```

16.4 Other containers

Tk provides just a few other basic containers, here we describe paned windows and notebooks.

Paned Windows

A paned window is constructed by the function `ttkpanedwindow`. The primary option, outside of setting the requested width or height with `width` and `height`, is `orient`, which takes a value of "vertical" (the default) or "horizontal". This specifies how the children are stacked, and is opposite how the sash is drawn.

The returned object can be used as a parent container, although one does not use the geometry managers to manage them. Instead, the `add` command is used. For example:

```
w <- tkoplevel(); tkwm.title(w, "Paned window example")
pw <- ttkpanedwindow(w, orient="horizontal")
tkpack(pw, expand=TRUE, fill="both")
left <- ttklabel(pw, text="left")
right <- ttklabel(pw, text="right")
#
tkadd(pw, left, weight=1)
tkadd(pw, right, weight=2)
```

When resizing which child gets the space is determined by the associated weight, specified as an integer. The default uses even weights. Unlike GTK+ more than two children are allowed.

Forget The subcommand `ttkpanedwindow forget` can be used to unmanage a child component. For the paned window, we have no convenience function, so we call as follows:

```
tcl(pw, "forget", right)
tkadd(pw, right, weight=2) ## add back
```

Sash position The sash between two children can be adjusted through the subcommand `ttkpanedwindow sashpos`. The index of the sash needs specifying, as there can be more than one. Counting starts at 0. The value for `sashpos` is in terms of pixel width (or height) of the paned window. The width can be returned as follows:

```
tcl(pw, "sashpos", 0, 150)
```

```
<Tcl> 59
```

```
as.integer(tkwininfo("width", pw)) # or "height"
```

```
[1] 71
```

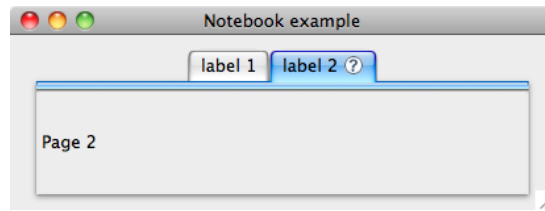


Figure 16.6: A basic notebook under Mac OS X

Notebooks

The `ttknotebook` constructor returns a notebook object. In Tk the object itself, becomes a command with the subcommands being important. There are no convenience functions for these, so we will use the `tcl` function directly.

Notebook pages can be added through the `ttknotebook add` subcommand or inserted after a page through the `ttknotebook insert` subcommand. The latter requires a tab ID to be specified, as described below. The tab label is configured similarly to `ttklabel` through the options `text` and the optional `image`, which if given has its placement determined by `compound`. The placement of the child component within the notebook page is manipulated similarly as `tkgrid` through the `sticky` option with values specified through compass points. Extra padding around the child can be added with the `padding` option. Typically, the child components would be containers to hold more complicated layouts.

Tab identifiers Many of the commands for a notebook require a specification of a desired tab. This can be given by index, starting at 0; by the values "current" or "end"; by the child object added to the tab, either as an R object or an ID; or in terms of *x-y* coordinates in the form "@x,y" (likely found through a binding).

To illustrate, if `nb` is a `ttknotebook` object, then these commands would add pages (cf. Figure 16.6):

```
iconFile <- system.file("images",paste("help","gif",sep="."),
                        package="gWidgets")
iconName <- "::tcl::helpIcon"
QT <- tkimage.create("photo", iconName, file = iconFile)
#
l2 <- ttklabel(nb, text="Page 2")
tkadd(nb, l2, sticky="nswe", text="label 2",
      image=iconName, compound="right")
## put l1 first (tabID is 0), use tkinsert
l1 <- ttklabel(nb, text="Page 1")
tkinsert(nb, 0, l1, sticky="nswe", text="label 1")
```

There are several useful subcommands to extract information from the notebook object. For instance, `index` to return the page index (0-based), `tabs` to return the page IDs, `select` to select the displayed page, and `forget` to remove a page from the notebook. Except for `tabs`, these require a specification of a tab ID.

```
tcl(nb, "index", "current")          # current page for tabID

<Tcl> 1

length(as.character(tcl(nb,"tabs"))) # number of pages

[1] 2

tcl(nb, "select", 0)                 # select viewable page by index
tcl(nb, "forget", 11)                # forget removes page from notebook
tcl(nb, "add", 11)                   # can be managed again.
```

The notebook state can be manipulated through the keyboard, provided traversal is enabled. This can be done through

```
QT <- tcl("ttk::notebook::enableTraversal", nb)
```

If enabled, the shortcuts such as control-tab to move to the next tab are implemented. If new pages are added or inserted with the option `underline`, which takes an integer value (0-based) specifying which character in the label is underlined, then a keyboard accelerator is added for that letter.

Bindings Beyond the usual events, the notebook widget also generates a `<<NotebookTabChanged>>` virtual event after a new tab is selected.

The notebook container in Tk has a few limitations. For instance, there is no graceful management of too many tabs, as there is with GTK+, as well there is no easy way to implement close buttons as an icon, as in Qt.

Tcl Tk: Widgets

Tk has widgets for the common GUI controls. As mentioned in Chapter 15 – where we illustrated both buttons and labels – the constructors for these widgets call the function `tkwidget` which calls the appropriate Tk command and adds in extra information including an ID and an environment. As with labels and buttons, one primarily uses `tkconfigure` and `tkcget` to set and get properties of the widget when a Tcl variable is not used to store the data for the widget.

17.1 Selection Widgets

This section covers the many different ways to present data for the user to select a value. The widgets can use Tcl variables to refer to the value that is displayed or that the user selects. Recall, these were constructed through `tclVar` and manipulated through `tclvalue`. For example, a logical value can be stored as

```
value <- tclVar(TRUE)
tclvalue(value) <- FALSE
tclvalue(value)
```

```
[1] "0"
```

As `tclvalue` coerces the logical into the character string "0" or "1", some coercion may be desired.

Checkbutton

The `ttkcheckbutton` constructor returns a check button object. The checkbutton's value (TRUE or FALSE) is linked to a Tcl variable which can be specified using a logical value. The checkbutton label can also be specified through a Tcl variable using the `textvariable` option. Alternately, as with the `ttklabel` constructor, the label can be specified through the `text` option.

This allows one to specify an image as well and arrange its display, as is done with `ttklabel`, using the `compound` option.

The `command` argument is used at construction time to specify a callback when the button is clicked. The callback is called when the state toggles, so often a callback considers the state of the widget before proceeding. To add a callback with `tkbind` use `<ButtonRelease-1>`, as the callback for the event `<Button-1>` is called before the variable is updated.

For example, if `f` is a frame, we can create a new check button with the following:

```
value <- tclVar(TRUE)
callback <- function() print(tclvalue(value))      # uses global
labelVar <- tclVar("check button label")
cb <- ttkcheckbutton(f, variable=value,
                    textvariable=labelVar, command=callback)
tkpack(cb)
```

To avoid using a global variable is not trivial here. There is no easy way to pass user data through to the callback, and there is no easy way to get the R object from the values passed through the `%` substitution values. The variable holding the value can be found through

```
tkcget(cb, "variable"=NULL)
```

```
<Tcl> ::RTcl3
```

But manipulating that is difficult. A more general strategy within R would be to use a function closure to encapsulate the variables or an environment to store the global values.

Radio Buttons

Radiobuttons are checkbuttons linked through a shared Tcl variable. Each button is constructed through the `ttkradiobutton` constructor. Each button has a value and a label, which need not be the same. The variable refers to the value. As with labels, the radio button labels may be specified through a text variable or the `text` option, in which case, as with a `ttklabel`, an image may also be incorporated through the `image` and `compound` options. In Tk the placement of the buttons is managed by the programmer.

This small example shows how radio buttons could be used for selection of an alternative hypothesis, assuming `f` is a parent container.

```
values <- c("less", "greater", "two.sided")
var <- tclVar(values[3])      # initial value
callback <- function() print(tclvalue(var))
sapply(values, function(i) {
  rb <- ttkradiobutton(f, text=i, variable=var,
                      value=i, command=callback)
```

```
tkpack(rb, side="top", anchor="w")
})
```

```
$less
```

```
$greater
```

```
$two.sided
```

Comboboxes

The `ttkcombobox` constructor returns a combobox object to select from a list of values, or with the appropriate option, allowing the user to specify a value. Like radiobuttons and checkbuttons, the value of the combobox can be specified using a Tcl variable to the option `textvariable`, making the getting and setting of the displayed value straightforward. The possible values to select from are specified as a character vector through the `values` option. (This may require one to coerce the results to the desired class.) Unlike GTK+ and Qt there is no option to include images in the displayed text. One can adjust the alignment through the `justify` options. By default, a user can add in additional values through the entry widget part of the combobox. The `state` option controls this, with the default "normal" and the value "readonly" as an alternative.

To illustrate, again suppose `f` is a parent container. Then we begin by defining some values to choose from and a Tcl variable.

```
values <- rownames(mtcars)
var <- tclVar(values[1])           # initial value
```

The constructor call is as follows:

```
cb <- ttkcombobox(f,
                  values=values,
                  textvariable=var,
                  state="normal", # or "readonly"
                  justify="left")
tkpack(cb)
```

The possible values the user can select from can be configured after construction through the `values` option:

```
tkconfigure(cb, values=tolower(values))
```

Setting the value Setting values can be done through the Tcl variable, or by value or index (0-based) using the `ttkcombobox set` sub command through `tkset` or the `ttkcombobox current` sub command.

```
tclvalue(var) <- values[2]      # using tcl variable
tkset(cb, values[4])           # by value
tcl(cb, "current", 4)          # by index
```

Getting the value One can retrieve the selected object in various ways: from the Tcl variable. Additionally, the *ttkcombobox* *get* subcommand can be used through *tkget*.

```
tclvalue(var)                  # TCL variable
```

```
[1] "hornet sportabout"
```

```
tkget(cb)                      # get subcommand
```

```
<Tcl> hornet sportabout
```

```
tcl(cb, "current")            # 0-based index
```

```
<Tcl> 4
```

Events The virtual event `<<ComboboxSelected>>` occurs with selection. When the combobox may be edited, a user may expect some action when the return key is pressed. This triggers a `<Return>` event. To bind to this event, one can do something like the following:

```
tkbind(cb, "<Return>", function(W) {
  val <- tkget(W)
  cat(as.character(val), "\n")
})
```

For editable comboboxes, the widget also supports some of the *ttkentry* commands discussed in Section 17.2.

Scale widgets

The *ttkscale* constructor to produce a themable scale (slider) control is missing¹. You can define your own:

```
ttkscale <- function(parent, ...) tkwidget(parent, "ttk::scale", ...)
```

The orientation is set through the option *orient* taking values of "horizontal" (the default) or "vertical". For sizing the slider, the *length* option is available. To set the range, the basic options are *from* and *to*. There is no *by* option as of Tk 8.5. The constructor *ttkscale*, for a non-themable slider, has the option *resolution* to set that. The *variable* option is used for specifying

¹As of R 2.11.0

a Tcl variable to record the value of the slider. Otherwise the `value` option is available. The `tkget` and `tkset` function (using the `ttkscale` `get` and `ttkscale` `set` sub commands) can be used to get and set the value shown. They are used in the same manner as the same-named subcommands for a combobox. Again, the `command` option can be used to specify a callback for when the slider is manipulated by the user.

Spinboxes

In Tk version 8.5 there is no themable spinbox widget. In Tk the `spinbox` command produces a non-themable spinbox. Again, there is no direct `tkspinbox` constructor, but one can be defined with:

```
tkspinbox <- function(parent, ...)
  tkwidget(parent, "tk::spinbox", ...)
```

The non-themable widgets have many more options than the themable ones, as style properties can be set on a per-widget basis. We won't discuss those here. The spinbox can be used to select from a sequence of numeric values or a vector of character values.

The basic options to set the range for a numeric spinbox are `from`, `to`, and `increment`. The `textvariable` option can be used to link the spinbox to a Tcl variable. As usual, this allows the user to easily get and set the value displayed. Otherwise, the `tkget` and `tkset` functions may be used for these tasks. The option `state` can be used to specify whether the user can enter values, the default of "normal"; not edit the value, but simply select one of the given values ("readonly"), or not select a value ("disabled").

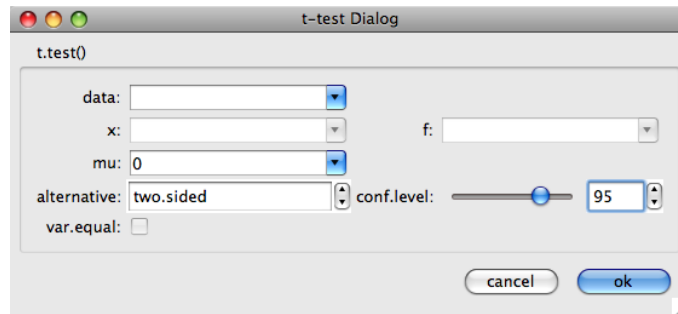
In Tk, spinboxes can also be used to select from a list of text values. These are specified through the `values` option. For the latter, the `wrap` option, as in `wrap=TRUE`, can be used to wrap around the list of values when the end is reached. As with a combobox, when the Tk spinbox displays character data and is in the "normal" state, the widget can be controlled like the entry widget of Section 17.2.

Example 17.1: A GUI for `t.test`

This example illustrates how the basic widgets can be combined to make a dialog for gathering information to run a *t*-test. A realization is shown in Figure 17.1.

We will use a data store to hold the values to be passed to `t.test`. For the data store, we use an environment to hold Tcl variables.

```
### Data model
e <- new.env()
e$x <- tclVar(""); e$f <- tclVar(""); e$data <- tclVar("")
e$mu <- tclVar(0); e$alternative <- tclVar("two.sided")
e$conf.level <- tclVar(95); e$var.equal <- tclVar(FALSE)
```

Figure 17.1: A dialog to collect values for a t test.

Our layout is basic. Here we pack a label frame into the window to give the dialog a nicer look. We will use the `tkgrid` geometry manager below.

```
lf <- ttklabelframe(f, text="t.test()", padding=10)
tkpack(lf, expand=TRUE, fill="both", padx=5, pady=5)
```

This next function simplifies the task of adding a label.

```
putLabel <- function(parent, text, row, column) {
  label <- ttklabel(parent, text=text)
  tkgrid(label, row=row, column=column, sticky="e")
}
```

Our first widget will be one to select a data frame. For this, a combobox is used, although if a large number of data frames are a possibility, a different widget may be better suited. The `getDfs` function is not shown, but simply returns the names of all data frames in the global environment. Also not shown are two similar calls to create comboboxes `xCombo` and `fCombo` which allow the user to specify parts of a formula.

```
putLabel(lf, "data:", 0, 0)
dataCombo <- ttkcombobox(lf, values=getDfs(), textvariable=e$data)
tkgrid(dataCombo, row=0, column=1, sticky="ew", padx=2)
tkfocus(dataCombo) # give focus
```

The combobox may not be the most natural widget to gather a numeric value for the mean when the data is continuous, but at this point we haven't quite yet discussed the `ttkentry` widget.

```
putLabel(lf, "mu:", 2, 0)
muCombo <- ttkcombobox(lf, values=c(""), textvariable=e$mu)
tkgrid(muCombo, row=2, column=1, sticky="ew", padx=2)
```

The selection of an alternative hypothesis is a natural choice for a combobox, but, as this alternative is available in `tcltk`, we use a spin box with `wrap=TRUE`.

```
putLabel(lf, "alternative:", 3, 0)
altCombo <- tkspinbox(lf, values=c("two.sided","less","greater"),
                      textvariable=e$alternative, wrap=TRUE)
tkgrid(altCombo, row=3, column=1, sticky="ew", padx=2)
```

Here we use two widgets to specify the confidence level. The slider is quicker to use, but less precise than the spinbox. By sharing a text variable, the widgets are automatically synchronized.

```
putLabel(lf, "conf.level:", 3, 2)
altFrame <- ttkframe(lf)
tkgrid(altFrame, row=3, column=3, columnspan=2,
       sticky="ew", padx=2)
altScale <- ttkScale(altFrame, from=75, to=100,
                    variable=e$conf.level)
tkpack(altScale, expand=TRUE, fill="y", side="left")
altSpin <- tkspinbox(altFrame, from=75, to=100, increment=1,
                    textvariable=e$conf.level, width=5)
tkpack(altSpin, side="left")
```

A checkbox is used to set the binary variable for `var.equal`

```
putLabel(lf, "var.equal:", 4, 0)
veCheck <- ttkcheckboxbutton(lf, variable=e$var.equal)
tkgrid(veCheck, row=4, column=1, stick="w", padx=2)
```

When assigning grid weights, we don't want the labels (columns 0 and 2) to expand the same way we want the other columns to do, so we assign different weights.

```
tkgrid.columnconfigure(lf, 0, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
tkgrid.columnconfigure(lf, 2, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
```

The dialog has two control buttons we wish to include.

```
bf <- ttkframe(f)
tkpack(bf, fill="x", padx=5, pady=5)
cancel <- ttkbutton(bf, text="cancel", command=function() {
  tcl("after","cancel",updateID$ID)
  tkdestroy(w)
})
ok <- ttkbutton(bf, text="ok")
tkpack(ttklabel(bf, text=" "), expand=TRUE, fill="y",
       side="left") # add a spring
tkpack(cancel, padx=6, side="left")
tkpack(ok, padx=6, side="left")
tkconfigure(ok, default="active")
```

The ok button is made to look active. As such we should bind to the button click and "Return" signals. First we define the callback. The `runTTest`

function is not shown, but is written to make good use of the structure of the data store.

```
okCallback <- function() {  
  l <- lapply(e, tclvalue)  
  runTTest(l)  
}  
tkbind(ok, "<Button-1>", okCallback)  
tkbind(w, "<Return>", okCallback)      # for active binding
```

At this point, our GUI is complete, but we would like to have it reflect any changes to the underlying R environment that effect its display. A such, we define a function, `updateUI`, which does two basic things: it searches for new data frames and it adjusts the controls depending on the current state.

```
updateUI <- function() {  
  tkconfigure(dataCombo, values=getDfs())  
  dfName <- tclvalue(e$data)  
  
  if(dfName == "") {  
    tkconfigure(xCombo, state="disabled")  
  } else {  
    df <- get(dfName, envir=.GlobalEnv)  
    tkconfigure(xCombo, state="normal", values=getNumericVars(df))  
    if(! tclvalue(e$x) %in% getNumericVars(df))  
      tclvalue(e$x) <- ""  
  
    tkconfigure(fCombo, values=getTwoLevelFactor(df))  
    if(! tclvalue(e$f) %in% getTwoLevelFactor(df))  
      tclvalue(e$f) <- ""  
  }  
  
  tkconfigure(fCombo, state=  
    ifelse(tclvalue(e$x) == "", "disabled", "normal"))  
  
  if(tclvalue(e$f) == "") {  
    tkconfigure(muCombo, state="normal")  
    tkconfigure(veCheck, state="disabled")  
  } else {  
    tclvalue(e$mu) <- 0  
    tkconfigure(muCombo, state="disabled")  
    tkconfigure(veCheck, state="normal")  
  }  
}
```

We use the `after` command to repeat a function call every so often. We also define a flag to stop the polling if desired. When polling, we make sure to test for existence of the parent window.

```
updateID <- new.env()
```



```

updateID$flag <- TRUE
updateID$ID <- NA
repeatFun <- function() {
  if(updateID$flag && as.logical(tkwininfo("exists",w))) {
    updateUI()
    updateID$ID <- tcl("after", 1000, repeatFun)
  }
}
repeatFun()

```

17.2 Text widgets

Tk provides both single- and multi-line text entry widgets. The section describes both and introduces scrollbars which are often desired for multi-line text entry.

Entry Widgets

The `ttkentry` constructor provides a single line text entry widget. The widget can be associated with a Tcl variable at construction to facilitate getting and setting the displayed values through its argument `textvariable`. The width of the widget can be adjusted at construction time through the `width` argument. This takes a value for the number of characters to be displayed, assuming average-width characters. The text alignment can be set through the `justify` argument taking values of "left" (the default), "right" and "center". For gathering passwords, the argument `show` can be used, such as with `show="*"`, to show asterisks in place of all the characters.

The following constructs a basic example

```

eVar <- tclVar("initial value")
e <- ttkentry(w, textvariable=eVar)
tkpack(e)

```

We can get and set values using the Tcl variable.

```
tclvalue(eVar)
```

```
[1] "initial value"
```

```
tclvalue(eVar) <- "set value"
```

The `get` command can also be used.

```
tkget(e)
```

```
<Tcl> set value
```

Indices The entry widget uses an index to record the different positions within the entry box. This index can be a number (0-based), an *x*-coordinate of the value (@x), the values "end" and "insert" to refer to the end of the current text and the insert as set through the keyboard or mouse. The mouse can also be used to make a selection. In this case the indices "sel.first" and "sel.last" describe the selection.

With indices, we can insert text with the *ttkentry* insert command

```
tkinsert(e, "end", "new text")
```

Or, we can delete a range of text, in this case the first 4 characters, using *ttkentry* delete. The first value is the left most index to delete (0-based), the second value the index to the right of the last value deleted.

```
tkdelete(e, 0, 4) # e.g., a b c d e f -text
```

The *ttkentry* icursor command can be used to set the cursor position to the specified index.

```
tkicursor(e, 0) # move to beginning
```

Finally, we note that the selection can be adjusted using the *ttkentry* selection range subcommand. This takes two indices. Like delete, the first index specifies the first character of the selection, the second indicates the character to the right of the selection boundary. The following example would select all the text.

```
tkselection.range(e, 0, "end")
```

The *ttkentry* selection clear subcommand clears the selection and *ttkentry* selection present signals if a selection is currently made.

Events Several useful events include <KeyPress> and <KeyRelease> for a key presses and <FocusIn> and <FocusOut> for focus events.

Example 17.2: Using validation for dates

There is no native calendar widget in *tcltk*. This example shows how one can use the validation framework for entry widgets to check that user-entered dates conform to an expected format.

Validation happens in a few steps. A validation command is assigned to some event. This call can come in two forms. Prevalidation is when a change is validated prior to being committed, for example when each key is pressed. Revalidation is when the value is checked after it is sent to be committed, say when the entry widget loses focus or the enter key is pressed.

When a validation command is called it should check whether the current state of the entry widget is valid or not. If valid, it returns a value of TRUE and FALSE otherwise. These need to be Tcl Boolean values, so in the following, the

command `tcl("eval","TRUE")` (or `tcl("eval", "FALSE")`) is used. If the validation command returns `FALSE`, then a subsequent call to the specified invalidation command is made.

For each callback, a number of substitution values are possible, in addition to the standard ones such as `W` to refer to the widget. These are: `d` for the type of validation being done: 1 for insert prevalidation, 0 for delete prevalidation, or -1 for revalidation; `i` for the index of the string to be inserted or deleted or -1; `P` for the new value if the edit is accepted (in prevalidation) or the current value in revalidation; `s` for the value prior to editing; `S` for the string being inserted or deleted, `v` for the current value of `validate` and `V` for the condition that triggered the callback.

In the following callback definition we use `W` so that we can change the entry text color to black and format the data in a standard manner and `P` to get the entry widget's value just prior to validations.

To begin, we define some patterns for acceptable date formats.

```
datePatterns <- c()
for(i in list(c("%m","%d","%Y"),      # U.S. style
              c("%m","%d","%y"))) {
  for(j in c("/", "-"," "))
    datePatterns[length(datePatterns)+1] <-
      paste(i,sep="", collapse=j)
}
```

Our callbacks set the color to black or red, depending on whether we have a valid date. First our validation command.

```
isValidDate <- function(W, P) { # P is the current value
  for(i in datePatterns) {
    date <- try( as.Date(P, format=i), silent=TRUE)
    if(!inherits(date, "try-error") && !is.na(date)) {
      tkconfigure(W,foreground="black") # consult style?
      tkdelete(W,"0","end")
      tkinsert(W,0, format(date, format="%m/%d/%y"))
      return(tcl("expr","TRUE"))
    }
  }
  return(tcl("expr","FALSE"))
}
```

This is our invalid command.

```
indicateInvalidDate <- function(W) {
  tkconfigure(W,foreground="red")
  tcl("expr","TRUE")
}
```

The `validate` argument is used to specify when the validation command should be called. This can be a value of `"none"` for validation when called

through the validation command; "key" for each key press; "focusin" for when the widget receives the focus; "focusout" for when it loses focus; "focus" for both of the previous; and "all" for any of the previous. We use "focusout" below, so also give a button widget so that the focus can be set elsewhere. (As usual, *f* is a parent frame.)

```
e <- ttkentry(f, validate="focusout",
              validatecommand=isValidDate,
              invalidcommand=indicateInvalidDate)
tkpack(e, side="left")
b <- ttkbutton(f, text="click")           # something to focus on
tkpack(b, side="bottom")
```

Scrollbars

Tk has several scrollable widgets – those that use scrollbars. Widgets which accept a scrollbar (without too many extra steps) have the options `xscrollcommand` and `yscrollcommand`. To use scrollbars in `tcltk` requires two steps: the scrollbars must be constructed and bound to some widget, and that widget must be told it has a scrollbar. This way changes to the widget can update the scrollbar and vice versa. Suppose, *parent* is a container and *widget* has these options, then the following will set up both horizontal and vertical scrollbars.

The scrollbars are defined as follows using the `orient` option and a command of the following form.

```
xscr <- ttkscrollbar(parent, orient="horizontal",
                    command=function(...) tkxview(widget, ...))
yscr <- ttkscrollbar(parent, orient="vertical",
                    command=function(...) tkxview(widget, ...))
```

The view commands set what part of the widget is being shown.

To link the widget back to the scrollbar, the `set` command is used in a callback to the scroll command. For this example we configure the options after the widget is constructed, but this can be done at the time of construction as well. Again, the command takes a standard form:

```
tkconfigure(widget,
            xscrollcommand=function(...) tkset(xscr,...),
            yscrollcommand=function(...) tkset(yscr,...))
```

Although scrollbars can appear anywhere, the conventional place is on the right and lower side of the parent. The following adds scrollbars using the grid manager. The combination of weights and stickiness below will have the scrollbars expand as expected if the window is resized.

```
tkgrid(widget, row=0, column=0, sticky="news")
tkgrid(yscr, row=0, column=1, sticky="ns")
tkgrid(xscr, row=1, column=0, sticky="ew")
```

```
tkgrid.columnconfigure(parent, 0, weight=1)
tkgrid.rowconfigure(parent, 0, weight=1)
```

Although a bit tedious, this gives the programmer some flexibility in arranging scrollbars. To avoid doing all this in the sequel, we turn the above into function `addScrollbars` (not shown).

Multi-line Text Widgets

The `tktext` widget creates a multi-line text editing widget. If constructed with no options but a parent container, the widget can have text entered into it by the user.

The text widget is not a themed widget, hence has numerous arguments to adjust its appearance. We mention a few here and leave the rest to be discovered in the manual page (along with much else). The argument `width` and `height` are there to set the initial size, with values specifying number of characters and number of lines (not pixels). The actual size is font dependent, with the default for 80 by 24 characters. The `wrap` argument, with a value from "none", "char", or "word", indicates if wrapping is to occur and if so, does it happen at any character or only a word boundary. The argument `undo` takes a logical value indicating if the undo mechanism should be used. If so, the subcommand `tktext edit` can be used to undo a change (or the control-z keyboard combination).

Indices As with the entry widget, several commands take indices to specify position within the text buffer. Only for the multi-line widget both a line and character are needed in some instances. These indices may be specified in many ways. One can use row and character numbers separated by a period in the pattern `line.char`. The line is 1-based, the column 0-based (e.g., 1.0 says start on the 1st row and first character). In general, one can specify any line number and character on that line, with the keyword `end` used to refer to the last character on the line. Text buffers may carry transient marks, in which case the use of this mark indicates the next character after the mark. Predefined marks include `end`, to specify the end of the buffer, `insert`, to track the insertion point in the text buffer were the user to begin typing, and `current`, which follows the character closest to the mouse position. As well, pieces of text may be tagged. The format `tag.first` and `tag.last` index the range of the tag `tag`. Marks and tags are described below. If the *x-y* position of the spot is known (through percent substitutions say) the index can be specified by position, as *x,y*.

Indices can also be adjusted relative to the above specifications. This adjustment can be by a number of characters (`chars`), index positions (`indices`) or lines. For example, `insert + 1 lines` refers to 1 line under the insert point. The values `linestart`, `lineend`, `wordstart` and `wordend` are also avail-

able. For instance, `insert linestart` is the beginning of the line from the insert point, while `end -1 wordstart` and `end - 1 chars wordend` refer to the beginning and ending of the last word in the buffer. (The end index refers to the character just after the new line so we go back 2 steps.)

Getting text The *tktext* `get` subcommand is used to retrieve the text in the buffer. Coercion to character should be done with `tclvalue` and not `as.character` to preserve the distinction between spaces and line breaks.

```
value <- tkget(t, "1.0", "end")  
as.character(value)           # wrong way
```

```
character(0)
```

```
tclvalue(value)
```

```
[1] "\n"
```

Inserting text Inserting text can be done through the *tktext* `insert` subcommand by specifying first the index then the text to add. One can use `\n` to add new lines.

```
tkinsert(t, "end", "more text\n new line")
```

Images and other windows can be added to a text buffer, but we do not discuss that here.

The buffer can have its contents cleared using `tkdelete`, as with `tkdelete(t, "0.0", "end")`.

Panning the buffer: tksee After text is inserted, the visible part of buffer may not be what is desired. The *tktext* `see` sub command is used to position the buffer on the specified index, its lone argument.

tags Tags are a means to assign a name to characters within the text buffer. Tags may be used to adjust the foreground, background and font properties of the tagged characters from those specified globally at the time of construction of the widget, or configured thereafter. Tags can be set when the text is inserted, as with

```
tkinsert(t, "end", "last words", "lastWords") # lastWords is tag
```

Tags can be set after the text is added through the *tktext* `tag add` subcommand using indices to specify location. The following marks the first word:

```
tktag.add(t, "firstWord", "1.0 wordstart", "1.0 wordend")
```

The *tktext* `tag configure` can be used to configure properties of the tagged characters, for example:

```
tktag.configure(t, "firstWord", foreground="red",
               font="helvetica 12 bold")
```

There are several other configuration options for a tag. A cryptic list can be produced by calling the subcommand *tktext* `tag configure` without a value for configuration.

selection The current selection, if any, is indicated by the `sel` tag, with `sel.first` and `sel.last` providing indices to refer to the selection. (Provided the option `exportSelection` was not modified.) These tags can be used with `tkget` to retrieve the currently selected text. An error will be thrown if there is no current selection. To check if there is a current selection, the following may be used:

```
hasSelection <- function(W) {
  ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
  length(ranges) > 1 || ranges != ""
}
```

The cut, copy and paste commands are implemented through the functions `tk_textCut`, `tk_textCopy` and `tk_textPaste`. Their lone argument is the text widget. These work with the current selection and insert point. For example to cut the current selection, one has

```
tcl("tk_textCut", t)
```

marks Tags mark characters within a buffer, marks denote positions within a buffer that can be modified. For example, the marks `insert` and `current` refer to the position of the cursor and the current position of the mouse. Such information can be used to provide context-sensitive popup menus, as in this code example:

```
popupContext <- function(W, x, y) {
  ## or use sprintf("@%s,%s", x, y) for "current"
  cur <- tkget(W, "current wordstart", "current wordend")
  cur <- tclvalue(cur)
  popupContextMenuFor(cur, x, y)      # some function
}
```

To assign a new mark, one uses the *tktext* `mark set` subcommand specifying a name and a position through an index. Marks refer to spaces within characters. The gravity of the mark can be left or right. When right (the default), new text inserted is to the left of the mark. For instance, to keep track of an initial insert point and the current one, the initial point (marked `leftlimit` below) can be marked with

```
tkmark.set(t,"leftlimit","insert")
tkmark.gravity(t,"leftlimit","left")    # keep onleft
tkinsert(t,"insert","new text")
tkget(t, "leftlimit", "insert")
```

<Tcl> new text

The use of the subcommand *tktext* mark gravity is done so that the mark attaches to the left-most character at the insert point. The rightmost one changes as more text is inserted, so would make a poor choice.

The edit command The subcommand *tktext* edit can be used to undo text. As well, it can be used to test if the buffer has been modified, as follows:

```
tcl(t, "edit", "undo")                # no output
tcl(t, "edit", "modified")            # 1 = TRUE
```

<Tcl> 1

Events The text widget has a few important events. The widget defines virtual events <<Modified>> and <<Selection>> indicating when the buffer is modified or the selection is changed. Like the single-line text widget, the events <KeyPress> and <KeyRelease> indicate key activity. The %-substitution *k* gives the keycode and *K* the key symbol as a string (*N* is the decimal number).

Example 17.3: Displaying commands in a text buffer

This example shows how a text buffer can be used to display the output of R commands, using an approach modified from Sweave.

```
## create formatting tags
tktag.configure(t, "commandTag", foreground="blue",
                font="courier 12 italic")
tktag.configure(t, "outputTag", font="courier 12")
tktag.configure(t, "errorTag", foreground="red",
                font="courier 12 bold")
```

The following function does the work of evaluating a command chunk then inserting the values into the text buffer, using the different markup tags specified above to indicate commands from output.

```
evalCmdChunk <- function(t, cmds) {

  cmdChunks <- try(parse(text=cmds), silent=TRUE)
  if(inherits(cmdChunks,"try-error")) {
    tkinsert(t, "end", "Error", "errorTag") # add tag for markup
  }
}
```



```

for(cmd in cmdChunks) {
  dcmd <- deparse(cmd, width.cutoff = 0.75 * getOption("width"))
  command <-
    paste(getOption("prompt"),
          paste(dcmd, collapse=paste("\n", getOption("continue")),
                sep="")),
          sep="", collapse="")
  tinsert(t, "end", command, "commandTag")
  tinsert(t, "end", "\n")
  ## output, should check for errors in eval!
  output <- capture.output(eval(cmd, envir=.GlobalEnv))
  output <- paste(output, collapse="\n")
  tinsert(t, "end", output, "outputTag")
  tinsert(t, "end", "\n")
}
}

```

We envision this as a piece of a larger GUI which generates the commands to evaluate. For this example though, we make a simple GUI.

```

w <- tktoplevel(); tkwm.title(w, "Text buffer example")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
t <- tktext(f, width=80, height = 24) # default size
addScrollbars(f, t)

```

This is how it can be used.

```
evalCmdChunk(t, "2 + 2; lm(mpg ~ wt, data=mtcars)")
```

17.3 Treeview widget

The themed treeview widget can be used to display rectangular data, like a data frame, or heirarchical data. The usage is similar for each beyond the need to indicate the heirarchical structure of a tree.

Rectangular data

Rectangular data has a row and column structure. In R, data frames are internally kept in terms of their columns which all must have the same type. The treeview widget is different, it stores all data as character data and one interacts with the data row by row.

The `ttktreeview` constructor creates the tree widget. There is no separate model for this widget, but there is a means to filter what is displayed. The argument `columns` is used to specify internal names for the columns and indicate the number of columns. A value of `1:n` will work here unless explicit names are desired. The argument `displaycolumns` is used to control which of

the columns are actually display. The default is "all", but a vector of indices or names can be given. The size of the widget is specified two different ways. The `height` argument is used to adjust the number of visible rows. The width of the widget is determined by the combined widths of each column, whose adjustments are mentioned later. The user may select one or more rows with the mouse, as controlled by the argument `selectmode`. Multiple rows may be selected with the default value of "extended", a restriction to a single row is specified with "browse", and no selection is possible if this is given as none. The treeview widget has an initial column for showing the tree-like aspect with the data. This column is referenced by #0. The `show` argument controls whether this column is shown. A value of "tree" leaves just this column shown, "headings" will show the other columns, but not the first, and the combined value of "tree headings" will display both (the default). Additionally, the treeview is a scrollable widget, so has the arguments `xscrollcommand` and `yscrollcommand` for specifying scrollbars.

If `f` is a frame, then the following call will create a widget with just one column showing 25 rows, like the older, non-themed, listbox widget of Tk.

```
tr <- ttktreeview(f,
                  columns=1,           # column identifier is "1"
                  show="headings",     # not "#0"
                  height=25)
addScrollbars(f, tr)                 # scrollbar function
```

Column properties Once the widget is constructed, its columns can be configured on a per-column basis. Columns can be referred to by the name specified through the `columns` argument or by number starting at 1 with "#0" referring to the tree column. The column headings can be set through the `ttktreeview` heading subcommand. The heading, similar to the button widget, can be text, an image or both. The text placement of the heading may be positioned through the `anchor` option. For example, this command will center the text heading of the first column:

```
tcl(tr, "heading", 1, text="Host", anchor="center")
```

The `ttktreeview` column subcommand can be used to adjust a column's properties including the size of the column. The option `width` is used to specify the pixel width of the column (the default is large); As the widget may be resized, one can specify the minimum column width through the option `minwidth`. When more space is allocated to the tree widget, than is requested by the columns, column with a TRUE value specified to the option `stretch` are resized to fill the available space. Within each column, the placement of each entry within a cell is controlled by the `anchor` option, using the compass points.

For example, this command will adjust properties of the lone column of `tr`:

```
tcl(tr, "column", 1, width=400, stretch=TRUE, anchor="w")
```

Adding values Values can be added to the widget through the *ttktreeview* *insert parent item [text] [values]* subcommand. This requires the specification of a parent (always "" for rectangular data) and an index for specifying the location of the new child amongst the previous children. The special value "end" indicates placement after all other children, as would a number larger than the number of children. A value of 0 or a negative value would put it at the beginning.

There are a number of options for each row. If column #0 is present, the text option is used to specify the text for the tree row and the option image can be given to specify an image to place to the left of the text value. For filling in the columns the values option is used. If there is a single column, like the current example, care needs to be taken that values separated by spaces are quoted (or in braces), otherwise, they will be split on spaces and treated like a vector of values truncated on the first one. Finally, we mention that tag option for insert that can be used to specify a tag for the inserted row. This allows the use of the subcommand *ttktreeview* tag configure to configure the foreground color, background color, font or image of an item.

In the example this is how we can add a list of possible CRAN mirrors to the treeview display.

```
x <- getCRANmirrors()
Host <- paste("'", x$Host, "'", sep="") # add quotes!
shade <- c("none", "gray")           # tag names
for(i in 1:length(Host))
  ID <- tkinset(tr, "", "end", values=Host[i],
               tag=shade[i % 2])      # none or gray
tktag.configure(tr, "gray", background="gray95") # shade rows
```

Item IDs Each row has a unique item ID generated by the widget when a row is added. The base ID is "" (why this is specified for the value of parent for rectangular data). For rectangular displays, the list of all IDs may be found through the *ttktreeview* children sub command, which we will describe in the next section. Here we see it used to find the children of the root. As well, we show how the *ttktreeview* index command returns the row index.

```
children <- tcl(tr, "children", "")
(children <- head(as.character(children))) # as.character
```

```
[1] "I001" "I002" "I003" "I004" "I005" "I006"
```

```
sapply(children, function(i) tclvalue(tkindex(tr, i)))
```

```
I001 I002 I003 I004 I005 I006
"0"  "1"  "2"  "3"  "4"  "5"
```

Retreiving values The *ttktreeview* *item* subcommand can be used to get the values and other properties stored for each row. One specifies the item and the corresponding option:

```
x <- tcl(tr, "item", children[1], "-values") # no tkitem
as.character(x)
```

```
[1] "Patan.com.ar, Buenos Aires"
```

The value returned from the *item* command can be difficult to parse, as Tcl introduces braces for grouping. The coercion through *as.character* works much better at extracting the individual columns. A possible alternative to using the *item* command, is to instead keep the original data frame and use the index of the item to extract the value from the original.

Moving and deleting items The *ttktreeview* *move* subcommand can be used to replace a child. As with the *insert* command, a parent and an index for where the new child is to go among the existing children is given. The item to be moved is referred to by its ID. The *ttktreeview* *delete* and *ttktreeview* *detach* can be used to remove an item from the display, as specified by its ID. The latter command allows for the item to be reinserted at a later time.

Events and callbacks In addition to the keyboard events *<KeyPress>* and *<KeyRelease>* and the mouse events *<ButtonPress>*, *<ButtonRelease>* and *<Motion>*, the virtual event *<<TreeviewSelect>>* is generated when the selection changes. The current selection marks 0, 1 or more than 1 items if "extended" is given for the *selectmode* argument. The *ttktreeview* *selection* command will return the current selection. If converted to a string using *as.character* this will be a 0-length character vector, or a character vector of the selected item IDs. Further subcommands *set*, *add*, *remove*, and *toggle* can be used to adjust the selection programatically.

Within a key or mouse event callback, the selected column and row can be identified by position, as illustrated in this example callback.

```
callbackExample <- function(W, x, y) {
  col <- as.character(tkidentify(W, "column", x, y))
  row <- as.character(tkidentify(W, "row", x, y))
  ## do something ...
}
```

Example 17.4: Filtering a table

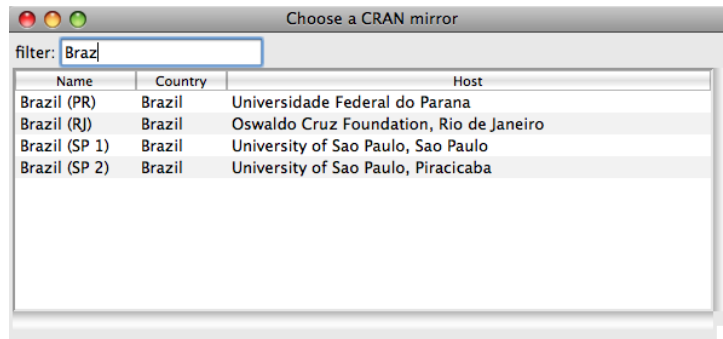


Figure 17.2: Using `ttktreeview` to show various CRAN sites. This illustration adds a search-like box to filter what repositories are displayed for selection.

We illustrate the above with a slightly improved GUI for selecting a CRAN mirror. This adds in a text box to filter the possibly large display of items to avoid scrolling through a long list.

```
df <- getCRANmirrors()[, c(1,2,5,4)]
```

We use a text entry widget to allow the user to filter the values in the display as the user types.

```
f0 <- ttkframe(f); tkpack(f0, fill="x")
l <- ttklabel(f0, text="filter:"); tkpack(l, side="left")
filterVar <- tclVar("")
filterEntry <- ttkentry(f0, textvariable=filterVar)
tkpack(filterEntry, side="left")
```

The treeview will only show the first three columns of the data frame, although we store the fourth which contains the URL.

```
f1 <- ttkframe(f); tkpack(f1, expand=TRUE, fill="both")
tr <- ttktreeview(f1, columns=1:ncol(df),
                  displaycolumns = 1:(ncol(df) - 1),
                  show = "headings",      # not "tree"
                  selectmode = "browse") # single selection
addScrollbars(f1, tr)
```

We configure the column widths and titles as follows:

```
widths <- c(100, 75, 400)          # hard coded
nms <- names(df)
for(i in 1:3) {
  tcl(tr, "heading", i, text=nms[i])
  tcl(tr, "column", i, width=widths[i], stretch=TRUE, anchor="w")
}
```

This following helper function is used to fill in the widget with values from a data frame.

```
fillTable <- function(tr, df) {  
  children <- as.character(tcl(tr, "children", ""))  
  for(i in children) tcl(tr, "delete", i)  
  shade <- c("none", "gray")  
  for(i in seq_len(nrow(df)))  
    tcl(tr, "insert", "", "end", tag=shade[i %% 2], text="",  
        values=unlist(df[i,]))  
  tktag.configure(tr, "gray", background="gray95")  
}
```

The initial call populates the table from the entire data frame.

```
fillTable(tr, df)
```

The filter works by grepping the user input against the host value. We bind to `<KeyRelease>` (and not `<KeyPress>`) so we capture the last keystroke.

```
curInd <- 1:nrow(df)  
tkbind(filterEntry, "<KeyRelease>", function(W, K) {  
  val <- tclvalue(tkget(W))  
  possVals <- apply(df, 1, function(...) paste(..., collapse=" "))  
  ind <- grep(val, possVals)  
  if(length(ind) == 0) ind <- 1:nrow(df)  
  fillTable(tr, df[ind,])  
})
```

This binding is for capturing a user's selection through a double-click event. In the callback, we set the CRAN option then withdraw the window.

```
tkbind(tr, "<Double-Button-1>", function(W, x, y) {  
  sel <- as.character(tcl(W, "identify", "row", x, y))  
  vals <- tcl(W, "item", sel, "-values")  
  URL <- as.character(vals)[4] # not tclvalue  
  repos <- getOption("repos")  
  repos["CRAN"] <- gsub("/$", "", URL[1L])  
  options(repos = repos)  
  tkwm.withdraw(tkwininfo("toplevel", W))  
})
```

Editing cells of a table There is no native widget for editing the cells of tabular data, as is provided by the `edit` method for data frames. The `tktable` widget (<http://tktable.sourceforge.net/>) provides such an add-on to the base Tk. We don't illustrate its usage here, as we keep to the core set of functions provided by Tk. However, we note that the `gdf` function of `gWidgetstcltk` provides an example of how it can be used.

Heirarchical data

Specifying tree-like or heirarchical data is nearly identical to specifying rectangular data for the `ttktreeview` widget. The widget provides column #0 to display this extra structure. If an item, except the root, has children, a trigger icon to expand the tree is shown. This is in addition to any text and/or an icon that is specified. Children are displayed in an indented manner to indicate the level of ancestry they have relative to the root. To insert heirarchical data in to the widget the same `ttktreeview insert` subcommand is used, only instead of using the root item, "", as the parent item, one uses the item ID corresponding to the desired parent. If the option `open=TRUE` is specified to the `insert` subcommand, the children of the item will appear, if `FALSE`, the user can click the trigger icon to see the children. The programmer can use the `ttktreeview item` to configure this state. When the parent item is opened or closed, the virtual events `<<TreeviewOpen>>` and `<<TreeviewClose>>` will be signaled.

Traversal Once a tree is constructed, the programmer can traverse through the items using the subcommands `ttktreeview parent item` to get the ID for the parent of the item; `ttktreeview prev item` and `ttktreeview next item` to get the immediate siblings of the item; and `ttktreeview children item` to return the children of the item. Again, the latter one will produce a character vector of IDs for the children when coerced to character with `as.character`.

Example 17.5: Using the treeview widget to show an XML file

This example shows how to display the heirarchical structure of an XML document using the tree widget.

We use the XML library to parse a document from the internet. This example uses just a few functions from this library: The `(htmlTreeParse)` (similar to `xmlInternalTreeParse`) to parse the file, `xmlRoot` to find the base node, `xmlName` to get the name of a node, `xmlValue` to get an associated value, and `xmlChildren` to return any child nodes of a node.

```
library(XML)
fileName <- "http://www.omegahat.org/RFXML/shortIntro.html"
QT <- function(...) {} # quiet next call
doc <- htmlTreeParse(fileName, useInternalNodes=TRUE, error=QT)
root <- xmlRoot(doc)
```

Our GUI is primitive, with just a treeview instance added.

```
tr <- ttktreeview(f, displaycolumns="#all", columns=1)
addScrollbars(f, tr)
```

We configure our columns headers and set a minimum width below. Recall, the tree column is designated "#0".

```
tcl(tr, "heading", "#0", text="Name")
```

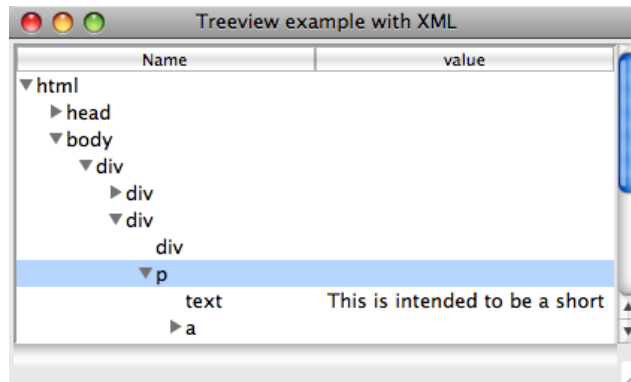


Figure 17.3: Illustration of using `ttktreeview` widget to show heirarchical data returned from parsing an HTML document with the XML package.

```
tcl(tr, "column", "#0", minwidth=20)
tcl(tr, "heading", 1, text="value")
tcl(tr, "column", 1, minwidth=20)
```

To map the tree-like structure of the XML document into the widget, we define the following function to recursively add to the treeview instance. We only add to the value column (through the `values` option) when the node does not have children. We use `do.call`, as a convenience, to avoid constructing two different calls to the `insert` subcommand. The `quoteIt` function used is not shown, but similar to `shQuote` only escaping with double quotes, as single quotes are treated differently by Tcl. (Otherwise the `ttktreeview` widget will split values on spaces.)

```
insertChild <- function(tr, node, parent="") {
  l <- list(tr, "insert", parent, "end", text=xmlName(node))
  children <- xmlChildren(node)
  if(length(children) == 0) {
    # add in values
    values <- paste(xmlValue(node), sep=" ", collapse=" ")
    values <- gsub("\n", " ", values) # treeview doesn't like
    values <- quoteIt(values)        # \n and spaces
    l$values <- values
  }
  treePath <- do.call("tcl", l)

  if(length(children)) # recurse
    for(i in children) insertChild(tr, i, treePath)
}
insertChild(tr, root)
```

At this point, the GUI will allow one to explore the structure of the XML file. We continue this example to show two things of general interest, but are

really artificial for this example.

Drag and drop First, we show how one might introduce drag and drop to rearrange the rows. We begin by defining two global variables that store the row that is being dragged and a flag to indicate if a drag event is ongoing.

```
.selectedID <- ""                                # globals
.dragging <- FALSE
```

We provide callbacks for three events: a mouse click, mouse motion and mouse release. This first callback sets the selected row on a mouse click.

```
tkbind(tr, "<Button-1>", function(W,x,y) {
    .selectedID <<- as.character(tcl(W, "identify","row", x, y))
})
```

The motion callback configures the cursor to indicate a drag event and sets the dragging flag. One might also put in code to highlight any drop areas.

```
tkbind(tr, "<B1-Motion>", function(W, x, y, X, Y) {
    tkconfigure(W, cursor="diamond_cross")
    .dragging <<-TRUE
})
```

When the mouse button is released we check that the widget we are over is indeed the tree widget. If so, we then move the rows. One can't move a parent to be a child of its own children, so we wrap the *ttktreeview* move subcommand within try. The move command places the new value as the first child of the item it is being dropped on. If a different action is desired, the "0" below would need to be modified.

```
tkbind(tr, "<ButtonRelease-1>", function(W, x, y, X, Y) {
    if(.dragging && .selectedID != "") {
        w = tkwininfo("containing", X, Y)
        if(as.character(w) == as.character(W)) {
            dropID <- as.character(tcl(W, "identify","row", x, y))
            try(tkmove(W, .selectedID, dropID, "0"), silent=TRUE)
        }
    }
    .dragging <<- FALSE; .selectedID <<- "" # reset
})
```

Walking the tree Our last item of general interest is a function that shows one way to walk the structure of the treeview widget to generate a list representing the structure of the data. A potential use of this might be to allow a user to rearrange an XML document through drag and drop. The subcommand *ttktreeview* children proves useful here, as it is used to identify the heirarchical structure. When there are children a recursive call is made.

```
treeToList <- function(tr) {  
  l <- list()  
  walkTree <- function(child, l) {  
    l$name <- tclvalue(tcl(tr,"item", child, "-text"))  
    l$value <- tclvalue(tcl(tr,"item", child, "-values"))  
    children <- as.character(tcl(tr, "children", child))  
    if(length(children)) {  
      l$children <- list()  
      for(i in children)  
        l$children[[i]] <- walkTree(i, list()) # recurse  
    }  
    return(l)  
  }  
  l <- walkTree("",l)  
  return(l)  
}
```

17.4 Menus

Menu bars and popup menus in Tk are constructed with `tkmenu`. The parent argument depends on what the menu is to do. A toplevel menu bar, such as appears at the top of a window has a toplevel window as its parent; a sub-menu of a menu bar uses the parent menu; and a popup menu uses a widget. The menu widget in Tk has an option to be “torn off.” This features was at one time common in GUIs, but now is rarely seen so it is recommended that this option be disabled. The `tearoff` option can be used at construction time to override the default behaviour. Otherwise, the following command will do so globally:

```
tcl("option","add","*tearOff", 0) # disable tearoff menus
```

A toplevel menu bar is attached to a top-level window using `tkconfigure` to set the menu property of the window. For the aqua Tk libraries for Mac OS X, this menu will appear on the top menu bar when the window has the focus. For other operating systems, it appears at the top of the window. For Mac OS X, a default menu bar with no relationship to your application will be shown if a menu is not provided for a toplevel window. Testing for native Mac OS X may be done via the following function:

```
usingMac <- function()  
  as.character(tcl("tk", "windowingsystem")) == "aqua"
```

The `tkpopup` function facilitates the creation of a popup menu. This function has arguments for the menu bar, and the position where the menu should be popped up. For example, the following code will bind a popup menu, `pmb` (yet to be defined), to the right click event for a button `b`. As Mac OS X may

not have a third mouse button, and when it does it refers to it differently, the callback is bound conditionally to different events.

```
doPopup <- function(X, Y) tkpopup(pmb, X, Y) # define call back
if (usingMac()) {
  tkbind(b, "<Button-2>", doPopup)      # right click
  tkbind(b, "<Control-1>", doPopup)    # Control + click
} else {
  tkbind(b, "<Button-3>", doPopup)
}
```

Adding submenus and action items *Menus* shows a heirarchical view of action items. Items are added to a menu through the *tkmenu* add subcommand. The nested structure of menus is achieved by specifying a *tkmenu* object as an item. The *tkmenu* add cascade subcommand is used for this. The option *label* is used to label the menu and the menu option to specify the sub-menu.

Grouping of similar items can be done through nesting, or on occasion through visual separation. The latter is implemented with the *tkmenu* add separator subcommand.

There are a few different types of action items that can be added.

An action item is one associated with a command. The simplest case is a label in the menu that activates a command when selected through the mouse. The *tkmenu* add command (through *tkadd*(widget, "command", ...)) allows one to specify a label, a command and optionally an image with a value for *compound* to adjust its layout. (Images are not shown in Mac OS X.) Action commands may possibly be called for different widgets, so the use of percent substitution is discouraged here. One can also specify that a keyboard accelerator be displayed through the option *accelerator*, but a separate callback must listen for this combination.

Action items may also be checkboxes. To create one, the subcommand *tkmenu* add checkbutton is used. The available arguments include *label* to specify the text, *variable* to specify a tcl variable to store the state, *onvalue* and *offvalue* to specify the state to the tcl variable, and *command* to specify a call back when the checked state is toggled. The initial state is set by the value in the Tcl variable.

Additionally, action items may be radiobutton groups. These are specified with the subcommand *tkmenu* add radiobutton. The *label* option is used to identify the entry, *variable* to set a text variable and to group the buttons that are added, and *command* to specify a command when that entry is selected.

Action items can also be placed after an item, rather than at the end using the *tkmenu* insert command *index* subcommand. The *index* may be specified numerically with 0 being the first item for a menu. More conveniently

the index can be determined by specifying a pattern to match the menu's labels.

Set state The `state` option is used to retrieve and set the current state of the a menu item. This value is typically `normal` or `disabled`, the latter to indicate the item is not available. The state can be set when the item is added or configured after that fact through the `tkmenu` `entryconfigure` command. This function needs the menu bar specified and the item specified as an index or pattern to match the labels.

Example 17.6: Simple menu example

This example shows how one might make a very simple code editor using a text-entry widget. We use the `svMisc` package, as it defines a few GUI helpers which we use.

```
library(svMisc)                                # for some helpers
showCmd <- function(cmd) writeLine(captureAll(Parse(cmd)))
```

We create a simple GUI with a top-level window containing the text entry widget.

```
w <- tktoplevel()
tkwm.title(w, "Simple code editor")
f <- ttkframe(w, padding=c(3,3,3,12));
tkpack(f, expand=TRUE, fill="both")
tb <- tktext(f, undo=TRUE)
addScrollbars(f, tb)
```

We create a toplevel menu bar, `mb`, and attach it to our toplevel window. Then we create a file and edit submenu.

```
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
editMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="Edit", menu=editMenu)
```

To these sub menu bars, we add action items. First a command to evaluate the contents of the buffer.

```
tkadd(fileMenu, "command", label="Evaluate buffer",
      command = function() {
        curVal <- tclvalue(tkget(tb, "1.0", "end"))
        showCmd(curVal)
      })
```

Then a command to evaluate just the current selection

```
tkadd(fileMenu, "command", label="Evaluate selection",
      state="disabled",
```

```

        command = function() {
            curSel <- tclvalue(tkget(tb, "sel.first", "sel.last"))
            showCmd(curSel)
        })

```

Finally, we end the file menu with a quit action.

```

tkadd(fileMenu, "separator")
tkadd(fileMenu, "command", label="Quit",
      command=function() tkdestroy(w))

```

The edit menu has an undo and redo item. For illustration purposes we add an icon to the undo item.

```

img <- system.file("images/up.gif", package="gWidgets")
QT <- tkimage.create("photo", "::img::undo",
                    file = img)
tkadd(editMenu, "command", label="Undo",
      image="::img::undo", compound="left",
      command = function() tcl(tb, "edit", "undo"))
tkadd(editMenu, "command", label="Redo",
      command = function() tcl(tb, "edit", "redo"))

```

We now define a function to update the user interface to reflect any changes.

```

updateUI <- function() {
    states <- c("disabled", "normal")
    ## selection
    hasSelection <- function(W) {
        ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
        length(ranges) > 1 || ranges != ""
    }
    ## by index
    tkentryconfigure(fileMenu, 1, state=states[hasSelection(tb) + 1])
    ## undo — if buffer modified, assume undo stack possible
    ## redo — no good check for redo
    canUndo <- function(W) as.logical(tcl(W, "edit", "modified"))
    tkentryconfigure(editMenu, "Undo", state=states[canUndo(tb) + 1])
    tkentryconfigure(editMenu, "Redo", state=states[canUndo(tb) + 1])
}

```

We now add an accelerator entry to the menubar and a binding to the top-level window for the keyboard shortcut.

```

if(usingMac()) {
    tkentryconfigure(editMenu, "Undo", accelerator="Cmd-z")
    tkbind(w, "<Option-z>", function() tcl(tb, "edit", "undo"))
} else {
    tkentryconfigure(editMenu, "Undo", accelerator="Control-u")
    tkbind(w, "<Control-u>", function() tcl(tb, "edit", "undo"))
}

```

To illustrate popup menus, we define one within our text widget that will grab all functions that complete the current word, using the `CompletePlus` function from the `svMisc` package to find the completions. The use of `current wordstart` and `current wordend` to find the word at the insertion point isn't quite right for R, as it stops at periods.

```
doPopup <- function(W, X, Y) {
  cur <- tclvalue(tkget(W, "current wordstart",
                        "current wordend"))
  tcl(W, "tag", "add", "popup", "current wordstart",
                        "current wordend")
  posVals <- head(CompletePlus(cur)[,1, drop=TRUE], n=20)
  if(length(posVals) > 1) {
    popup <- tkmenu(tb) # create menu for popup
    sapply(posVals, function(i) {
      tkadd(popup, "command", label=i, command = function() {
        tcl(W,"replace", "popup.first", "popup.last", i)
      })
    })
    tkpopup(popup, X, Y)
  }
}
```

For a popup, we set the appropriate binding for the underlying windowing system. For the second mouse button binding in OS X, we clear the clipboard. Otherwise the text will be pasted in, as this mouse action already has a default binding for the text widget.

```
if (!usingMac()) {
  tkbind(tb, "<Button-3>", doPopup)
} else {
  tkbind(tb, "<Button-2>", function(W,X,Y) {
    ## UNIX legacy re mouse-2 click for selection copy
    tcl("clipboard", "clear", displayof=W)
    doPopup(W,X,Y)
  }) # right click
  tkbind(tb, "<Control-1>", doPopup) # Control + click
}
```

17.5 Canvas Widget

The canvas widget provides an area to display lines, shapes, images and widgets. Methods exist to create, move and delete these objects, allowing the canvas widget to be the basis for creating interactive GUIs. The constructor `tkcanvas` for the widget, being a non-themable widget, has many arguments. We mention the standard ones `width`, `height`, and `background`. Additionally, the canvas is a scrollable widget, so has the corresponding arguments `xscrollcommand` and `yscrollcommand`.

The create command The subcommand *tkcanvas create type [options]* is used to add new items to the canvas. The options vary with the type of the item. The basic shape types that one can add are "line", "arc", "polygon", "rectangle", and "oval". Their options specify the size using *x* and *y* coordinates. Other options allow one to specify colors, etc. The complete list is covered in the canvas manual page, which we refer the reader to, as the description is lengthy. In the examples, we show how to use the "line" type to display a graph and how to use the "oval" type to add a point to a canvas. Additionally, one can add text items through the "text" type. The first options are the *x* and *y* coordinates and the *text* option specifies the text. Other standard text options are possible (e.g., *font*, *justify*, *anchor*).

The type can also be an image object or a widget (a window object). Images are added by specifying an *x* and *y* position, possibly an anchor position, and a value for the "image" option and optionally, for state dependent display, specifying "activeimage" and "disabledimage" values. The "state" option is used to specify the current state. Window objects are added similarly in terms of their positioning, along with options for "width" and "height". The window itself is added through the "window" option. An example shows how to add a frame widget.

Once created, a screenshot of the canvas can be created through the *tkcanvas postscript* subcommand, as in `tcl(canvas, "postscript", file="filename")`. To store the widget so that it can be recreated is not supported directly. Tcl code to do so can be found at <http://wiki.tcl.tk/9168>.

Items and tags The *tkcanvas.create* function returns an item ID. This can be used to refer to the item at a later stage. Optionally, tags can be used to group items into common groups. The "tags" option can be used with *tkcreate* when the item is created, or the *tkcanvas addtag* subcommand can be used. The call `tkaddtag(canvas, tagName, "withtag", item)` would add the tag "tagName" to the item returned by *tkcreate*. (The "withtag" is one of several search specifications.) As well, if one is adding a tag through a mouse click, the call `tkaddtag(W, "tagName", "closest", x, y)` could be used with *W*, *x* and *y* coming from percent substitutions. Tags can be deleted through the *tkcanvas dtag tag* subcommand.

There are several subcommands that can be called on items as specified by a tag or item ID. For example, the *tkcanvas itemcget* and *tkcanvas itemconfigure* subcommands allow one to get and set options for a given item. The *tkcanvas delete tag_or_ID* subcommand can be used to delete an item. Items can be moved through the *tkcanvas move tag_or_ID x y* subcommand, where *x* and *y* specify the horizontal and vertical shift in pixels. The subcommand *tkcanvas coords tag_or_ID [coordinates]* allows one to respecify the coordinates for which the item was defined, thereby allowing the possibility of moving or resizing the object. Additionally, the *tkcanvas scale* can

be used to rescale items. If items overlap each other, except for windows, an item can be raised to the top through the *tkcanvas* *raise item_or_ID* subcommand.

Bindings Bindings can be specified overall for the canvas, as usual, through *tkbind*. However, bindings can also be set on specific items through the subcommand *tkcanvas bind tag_or_ID event_function* which is aliased to *tkitembind*. This allows bindings to be placed on items sharing a tag name, without having the binding on all items. Only mouse, keyboard or virtual events can have such bindings.

Example 17.7: Using a canvas to make a scrollable frame

This example shows how to use a canvas widget to create a box container that scrolls when more items are added than will fit in the display area. The basic idea is that a frame is added to the canvas equipped with scrollbars using the *tkcanvas* *create window* subcommand. The binding to the *<Configure>* event updates the scrollregion of the canvas widget to include the entire canvas. This grows, as items are added to the frame. This is modified from an example found at <http://mail.python.org/pipermail/python-list/1999-June/005180.html>.

This constructor returns a box container that scrolls as more items are added. The parent passed in must use the grid manager for its children.

```
scrollableFrame <- function(parent, width= 300, height=300) {  
  canvasWidget <-  
    tkcanvas(parent,  
             borderwidth=0, highlightthickness=0,  
             background="#e3e3e3", # match themed widgets  
             width=width, height=height)  
  addScrollbars(parent, canvasWidget)  
  
  gp <- ttkframe(canvasWidget, padding=c(0,0,0,0))  
  gpID <- tkcreate(canvasWidget, "window", 0, 0, anchor="nw",  
                  window=gp)  
  
  tkbind(gp,"<Configure>",function() { # updates scrollregion  
    bbox <- tcl(canvasWidget, "bbox", "all")  
    tcl(canvasWidget,"config", scrollregion=bbox)  
  })  
  
  return(gp)  
}
```

To use it, we create a simple GUI as follows:

```
w <- tktoplevel()  
tkwm.title(w,"Scrollable frame example")
```

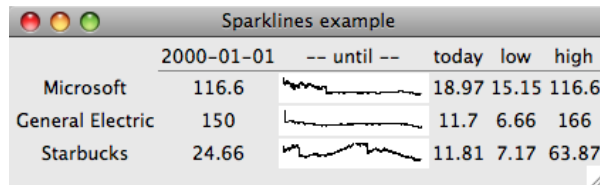



Figure 17.4: Example of embedding sparklines in a display organized using tkgrid. A tkcanvas widget is used to display the graph.

```
g <- ttkframe(w); tkpack(g, expand=TRUE, fill="both")
gp <- scrollableFrame(g, 300, 300)
```

To display a collection of available fonts requires a widget or container that could possibly show hundreds of similar values. The scrollable frame serves this purpose well (cf. Figure 15.2). The following shows how a label can be added to the frame whose font is the same as the label text. The available fonts are found from `tkfont.families` and the useful coercion to character by `as.character`.

```
fontFamilies <- as.character(tkfont.families())
## skip odd named ones
fontFamilies <- fontFamilies[grepl("^[[:alpha:]]", fontFamilies)]
for(i in 1:length(fontFamilies)) {
  fontName <- paste("tmp", i, sep="")
  try(tkfont.create(fontName, family=fontFamilies[i], size=14),
      silent=TRUE)
  l <- ttklabel(gp, text=fontFamilies[i], font=fontName)
  tkpack(l, side="top", anchor="w")
}
```

Example 17.8: Using canvas objects to show sparklines

Edward Tufte, in his book *Beautiful Evidence*?, advocates for the use of *sparklines* – small, intense, simple datawords – to show substantial amounts of data in a small visual space. This example shows how to use a `ttkcanvas` object to display a sparkline graph using a line object. The example also uses `tkgrid` to layout the information in a table. We could have spent more time on the formatting of the numeric values and factoring out the data download, but leave improvements as an exercise.

This function simply shortens our call to `ttklabel`. We use the global `f` (a `ttkframe`) as the parent.

```
mL <- function(label) {
  if(is.numeric(label))
    label <- format(label, digits=4)
  ttklabel(f, text=label) # save some typing
}
```

```
}
```

We begin by making the table header along with a toprule.

```
tkgrid(mL(""), mL("2000-01-01"), mL("-- until --"),
      mL("today"), mL("low"), mL("high"))
tkgrid(ttkseparator(f), row=1, column=1, columnspan=5, sticky="we")
```

This function adds a sparkline to the table. We use financial data in this example, as we can conveniently employ the `get.hist.quote` function from the `tseries` package to get interesting data.

```
addSparkLine <- function(label, symbol="MSFT") {
  width <- 100; height=15 # fix width, height
  y <- get.hist.quote(instrument=symbol, start="2000-01-01",
                     quote="C", provider="yahoo",
                     retclass="zoo")$Close
  min <- min(y); max <- max(y)
  start <- y[1]; end <- tail(y,n=1)
  rng <- range(y)

  sparkLineCanvas <- tkcanvas(f, width=width, height=height)
  x <- 0:(length(y)-1) * width/length(y)
  if(diff(rng) != 0) {
    y1 <- (y - rng[1])/diff(rng) * height
    y1 <- height - y1 # adjust to canvas coordinates
  } else {
    y1 <- height/2 + 0 * y
  }
  ## make line with: pathName create line x1 y1... xn yn
  l <- list(sparkLineCanvas,"create","line")
  sapply(1:length(x), function(i) {
    l[[2*i + 2]] <- x[i]
    l[[2*i + 3]] <- y1[i]
  })
  do.call("tcl",l)

  tkgrid(mL(label),mL(start), sparkLineCanvas,
        mL(end), mL(min), mL(max), pady=1)
}
```

We can then add some rows to the table as follows:

```
addSparkLine("Microsoft","MSFT")
addSparkLine("General Electric", "GE")
addSparkLine("Starbucks","SBUX")
```

Example 17.9: Capturing mouse movements

This example is a stripped-down version of the `tkcanvas.R` demo that accompanies the `tcltk` package. That example shows a scatterplot with regression

line. The user can move the points around and see the effect this has on the scatterplot. Here we focus on the moving of an object on a canvas widget. We assume we have such a widget in the variable `canvas`.

This following adds a single point to the canvas using an oval object. We add the "point" tag to this item, for later use. Clearly, this code could be modified to add more points.

```
x <- 200; y <- 150; r <- 6
item <- tkcreate(canvas, "oval", x - r, y - r, x + r, y + r,
                  width=1, outline="black",
                  fill="SkyBlue2")
tkaddtag(canvas, "point", "withtag", item)
```

In order to indicate to the user that a point is active, in some sense, the following changes the fill color of the point when the mouse is over the point. We add this binding using `tkitembind` so that it will apply to all point items and only the point items.

```
tkitembind(canvas, "point", "<Any-Enter>", function()
            tkitemconfigure(canvas, "current", fill="red"))
tkitembind(canvas, "point", "<Any-Leave>", function()
            tkitemconfigure(canvas, "current", fill="SkyBlue2"))
```

There are two key bindings needed for movement of an object. First, we tag the point item that gets selected when a mouse clicks on a point and update the last position of the currently selected point.

```
lastPos <- numeric(2) # global to track position
tagSelected <- function(W, x, y) {
  tkaddtag(W, "selected", "withtag", "current")
  tkitemraise(W, "current")
  lastPos <- as.numeric(c(x, y))
}
tkitembind(canvas, "point", "<Button-1>", tagSelected)
```

When the mouse moves, we use `tkmove` to have the currently selected point move too. This is done by tracking the differences between the last position recorded and the current position and moving accordingly.

```
moveSelected <- function(W, x, y) {
  pos <- as.numeric(c(x,y))
  tkmove(W, "selected", pos[1] - lastPos[1],
        pos[2] - lastPos[2])

  lastPos <- pos
}
tkbind(canvas, "<B1-Motion>", moveSelected)
```

A further binding, for the `<ButtonRelease-1>` event, would be added to do something after the point is released. In the original example, the old regression line is deleted, and a new one drawn. Here we simply delete the "selected" tag.

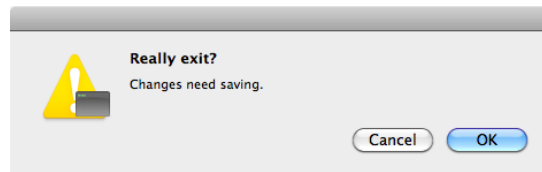


Figure 17.5: A basic modal dialog constructed by `tkmessageBox`.

```
tkbind(canvas, "<ButtonRelease-1>",  
       function(W) tkdtag(W, "selected"))
```

17.6 Dialogs

Modal dialogs

The `tkmessageBox` constructor can be used to create simple modal dialogs allowing a user to confirm an action. This replaces the older `tkdialog` dialogs. The `tkmessageBox` dialogs use the native toolkit if possible. The arguments `title`, `message` and `detail` are used to set the text for the dialog. The title may not appear for all operating systems. A `messageBox` dialog has an `icon` argument. The default icon is "info" but could also be one of "error", "question" or "warning". The buttons used are specified by the `type` argument with values of "ok", "okcancel", "retrycancel", "yesno", or "yesnocancel". When a button is clicked the dialog is destroyed and the button label returned as a value. The argument `parent` can be given to specify which window the dialog belongs to. Depending on the operating system this may be used when drawing the dialog.

A sample usage is:

```
tkmessageBox(title="Confirm", message="Really exit?",  
             detail="Changes need saving.",  
             icon="question", type="okcancel")
```

If the default modal dialog is not enough – for instance there is no means to gather user input – then a `toplevel` window can be made modal. The `tkwait.window` will cause a top-level window to be modal and `tkgrab.release` will return the interactivity for the window.

File and directory selection

Tk provides constructors for selecting a file, for selecting a directory or for specifying a filename when saving. These are implemented by `tkgetOpenFile`, `tkchooseDirectory`, and `tkgetSaveFile` respectively. Each of these can be called with no argument, and returns a `tclobj` that can be converted to a

character string with `tclvalue`. The value is empty when there is no selection made.

The dialog will appear related to a toplevel window if the argument `parent` is specified. The `initialdir` and `initialfile` can be used to specify the initial values in the dialog. The `defaulttextextension` argument can be used to specify a default extension for the file.

When browsing for files, it can be convenient to filter the available file types that can be selected. The `filetypes` argument is used for this task. However, the file types are specified using Tcl brace-notation, not R code. For example, to filter out various image types, one could have

```
tkgetOpenFile(filetypes = paste(
    "{{jpeg files} {.jpg .jpeg} }",
    "{{png files} {.png}}",
    "{{All files} {*}}", sep=" ") # needs space
```

Extending this is hopefully clear from the pattern above.

Example 17.10: A “File” menu

To illustrate, a simple example for a file menu could be:

```
w <- tktoplevel(); tkwm.title(w, "File menu example")
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
tkadd(fileMenu, "command", label="Source file...",
      command= function() {
        fname <- tkgetOpenFile(fileTypes=
                               "{{R files} {.R}} {{All files} *}")
        source(tclvalue(fname))
      })
tkadd(fileMenu, "command", label="Save workspace as...",
      command=function() {
        fname <- tkgetSaveFile(defaulttextextension="Rsave")
        save.image(file=tclvalue(fname))
      })
tkadd(fileMenu, "command", label="Set working directory...",
      command=function() {
        fname <- tkchooseDirectory()
        setwd(tclvalue(fname))
      })
```

Choosing a color

Tk provides the command `tk_chooseColor` to construct a dialog for selection of a color by RGB value. There are three optional arguments `initialcolor` to specify an initial color such as `"#efefef"`, `parent` to make the dialog a child

of a specified window and title to specify a title for the dialog. The return value is in hex-coded RGB quantiles. There is no constructor in `tcltk`, but one can use the dialog as follows:

```
w <- tktoplevel(); tkwm.title(w, "Select a color")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
colorWell <- tkcanvas(f, width=40, height=16,
                      background="#ee11aa",
                      highlightbackground="#ababab")

tkpack(colorWell)
tkbind(colorWell, "<Button-1>", function(W) {
  color <- tcl("tk_chooseColor", parent=w, title="Set box color")
  color <- tclvalue(color)
  if(nchar(color))
    tkconfigure(W, background = color)
})
```

Bibliography

- a. URL <http://www.tcl.tk/man/tcl8.5/>.
- b.
- Jeffrey Hobbs Brent B. Welch, Ken Jones. *Practical Programming in Tcl and Tk*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 2003.
- Peter Dalgaard. The R-Tcl/Tk interface. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*. URL <http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/index.html>. ISSN 1609-395X.
- Peter Dalgaard. A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31, September 2001. URL <http://CRAN.R-project.org/doc/Rnews/>.
- John Fox. Extending the R Commander by “plug-in” packages. *R News*, 7(3): 46–52, December 2007. URL <http://CRAN.R-project.org/doc/Rnews/>.
- Simon Urbanek. iwidgets - basic gui widgets for r. <http://www.rforge.net/iWidgets/index.html>.
- James Wettenhall. URL <http://bioinf.wehi.edu.au/~wettenhall/RTclTkExamples/>.