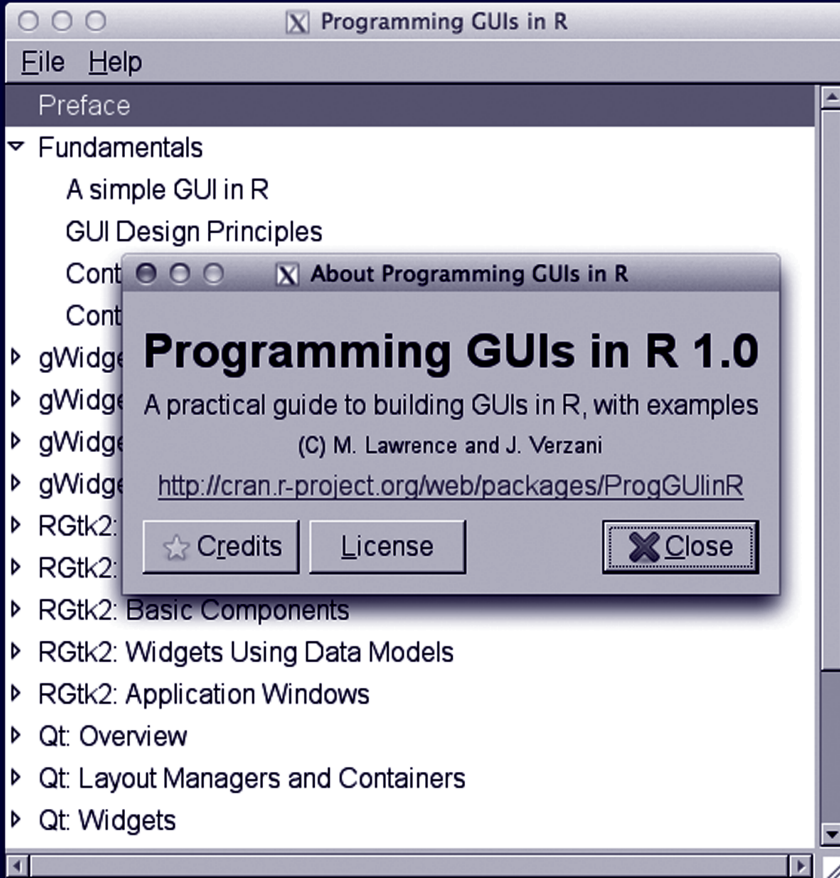


Programming Graphical User Interfaces in R



Michael F. Lawrence

John Verzani

Programming Graphical User Interfaces in R

Chapman & Hall/CRC

The R Series

Series Editors

John M. Chambers
Department of Statistics
Stanford University
Stanford, California, USA

Torsten Hothorn
Institut für Statistik
Ludwig-Maximilians-Universität
München, Germany

Duncan Temple Lang
Department of Statistics
University of California, Davis
Davis, California, USA

Hadley Wickham
Department of Statistics
Rice University
Houston, Texas, USA

Aims and Scope

This book series reflects the recent rapid growth in the development and application of R, the programming language and software environment for statistical computing and graphics. R is now widely used in academic research, education, and industry. It is constantly growing, with new versions of the core software released regularly and more than 2,600 packages available. It is difficult for the documentation to keep pace with the expansion of the software, and this vital book series provides a forum for the publication of books covering many aspects of the development and application of R.

The scope of the series is wide, covering three main threads:

- Applications of R to specific disciplines such as biology, epidemiology, genetics, engineering, finance, and the social sciences.
- Using R for the study of topics of statistical methodology, such as linear and mixed modeling, time series, Bayesian methods, and missing data.
- The development of R, including programming, building packages, and graphics.

The books will appeal to programmers and developers of R software, as well as applied statisticians and data analysts in many fields. The books will feature detailed worked examples and R code fully integrated into the text, ensuring their usefulness to researchers, practitioners and students.

Published Titles

Customer and Business Analytics: Applied Data Mining for Business Decision Making Using R, *Daniel S. Putler and Robert E. Krider*

Event History Analysis with R, *Göran Broström*

Programming Graphical User Interfaces with R, *Michael F. Lawrence and John Verzani*

R Graphics, Second Edition, *Paul Murrell*

Statistical Computing in C++ and R, *Randall L. Eubank and Ana Kupresanin*

The R Series

Programming Graphical User Interfaces in R

Michael F. Lawrence
John Verzani



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group an **informa** business

A CHAPMAN & HALL BOOK

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Nokia, the Nokia logo, Qt, and the Qt logo are trademarks of Nokia Corporation and/or its subsidiaries in Finland and other countries.

Linux is a ® trademark of Linus Torvalds in the United States, other countries or both.

Microsoft, Windows, XP, and the Windows logo are ® trademarks of Microsoft Corporation in the United States, other countries, or both.

Mac, Mac OS, OS X, and Time Machine are trademarks of Apple Inc., registered in the U.S. and other countries.

Java is a registered trademark of Oracle and/or its affiliates.

RStudio is a registered trademark of RStudio, Inc.

The TIBCO and Spotfire logos and Spotfire are trademarks or registered trademarks of TIBCO Software Inc.

SPSS is a registered trademarks of IBM.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2012 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20120503

International Standard Book Number-13: 978-1-4398-5683-3 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

Contents

Preface	xiii
1 The Fundamentals of Graphical User Interfaces	1
1.1 A simple GUI in R	1
1.2 GUI design principles	4
1.3 Controls	8
Choice of control	9
Presenting options	9
Checkboxes	9
Radio buttons	10
Combo boxes	10
List boxes	11
Sliders and spin buttons	11
Initiating an action	12
Buttons	12
Icons	12
Menu bars	13
Toolbars	13
Action objects	14
Modal dialogs	14
Message dialogs	14
File choosers	14
Displaying data	15
Tabular display	15
Tree widgets	15
Displaying and editing text	16
Single lines of text	16
Text-editing boxes	16
Guides and feedback	16
Labels	17
Status bars	17

	Tooltips	18
	Progress bars	18
1.4	Containers	18
	Top-level windows	19
	Tabbed notebooks	20
	Frames	20
	Expanding boxes	20
	Paned boxes	20
	Layout algorithms	21
	Box layout	21
	Grid layout	22
I	The gWidgets Package	23
2	gWidgets: Overview	25
2.1	Constructors	27
2.2	Methods	29
2.3	Event handlers	30
2.4	Dialogs	32
2.5	Installation	35
3	gWidgets: Container Widgets	37
3.1	Top-level windows	39
	A modal window	41
3.2	Box containers	42
	The ggroup container	42
	The gframe and gexpandgroup containers	45
3.3	Grid layout: the glayout container	46
3.4	Paned containers: the gpanedgroup container	47
3.5	Tabbed notebooks: the gnotebook container	48
4	gWidgets: Control Widgets	51
4.1	Buttons	51
4.2	Labels	53
	HTML text	53
	Status bars	53
	Icons and images	54
4.3	Text-editing controls	56
	Single-line, editable text	56
	Multiline, editable text	58
4.4	Selection controls	61
	Checkbox widget	61
	Radio buttons	62

	A group of checkboxes	63
	A combo box	64
	A slider control	67
	A spin button control	68
	Selecting from the file system	68
	Selecting a date	68
4.5	Display of tabular data	70
4.6	Display of hierarchical data	83
4.7	Actions, menus, and toolbars	86
	Toolbars	87
	Menu bars and pop-up menus	88
5	gWidgets: R-specific Widgets	91
5.1	A graphics device	91
5.2	A data frame editor	96
5.3	Workspace browser	97
5.4	Help browser	99
5.5	Command line widget	100
5.6	Simplifying creation of dialogs	100
II	The RGtk2 Package	101
6	RGtk2: Overview	103
6.1	Synopsis of the RGtk2R Package!RGtk2 API	104
6.2	Objects and classes	104
6.3	Constructors	105
6.4	Methods	108
6.5	Properties	109
6.6	Events and signals	110
6.7	Enumerated types and flags	112
6.8	The event loop	113
6.9	Importing a GUI from Glade	114
7	RGtk2: Windows, Containers, and Dialogs	115
7.1	Top-level windows	115
7.2	Layout containers	117
	Basics	117
	Widget size negotiation	118
	Box containers	119
	Alignment	123
7.3	Dialogs	124
	Message dialogs	124
	Custom dialogs	125

	File chooser	126
	Other choosers	127
	Print dialog	127
7.4	Special-purpose containers	128
	Framed containers	128
	Expandable containers	128
	Notebooks	128
	Scrollable windows	131
	Divided containers	132
	Tabular layout	133
8	RGtk2: Basic Components	137
8.1	Buttons	137
8.2	Static text and images	140
	Labels	140
	Images	142
	Stock icons	143
8.3	Input controls	143
	Text entry	143
	Check button	145
	Radio-button groups	146
	Combo boxes	147
	Sliders and spin buttons	149
8.4	Progress reporting	150
	Progress bars	150
	Spinners	151
8.5	Wizards	151
8.6	Embedding R graphics	156
8.7	Drag-and-drop	162
	Initiating a drag	163
	Handling drops	164
9	RGtk2: Widgets Using Data Models	165
9.1	Displaying tabular data	165
	Loading a data frame	165
	Displaying data as a list or table	166
	Accessing GtkTreeModel	169
	Selection	171
	Sorting	172
	Filtering	173
	Cell renderer details	175
9.2	Displaying hierarchical data	188
	Loading hierarchical data	188
	Displaying data as a tree	189

9.3	Model-based combo boxes	190
9.4	Text-entry widgets with completion	192
9.5	Sharing buffers between text entries	194
9.6	Text views	194
9.7	Text buffers	196
	Iterators	196
	Marks	198
	Tags	199
	Selection and the clipboard	200
	Inserting nontext items	200
10	RGtk2: Application Windows	205
10.1	Actions	205
10.2	Menus	207
	Menu bars	207
	Pop-up menus	209
10.3	Toolbars	210
10.4	Status reporting	213
	Status bars	213
	Info bars	214
10.5	Managing a complex user interface	215
11	Extending GObject Classes	221
III	The qtbase Package	225
12	Qt: Overview	227
12.1	The Qt library	227
12.2	An introductory example	228
12.3	Classes and objects	231
12.4	Methods and dispatch	233
12.5	Properties	234
12.6	Signals	235
12.7	Enumerations and flags	237
12.8	Extending Qt classes from R	237
	Defining a class	238
	Defining methods	238
	Defining signals and slots	239
	Defining properties	240
12.9	QWidget basics	243
	Fonts	244
	Styles	245
12.10	Importing a GUI from QtDesigner	247

13 Qt: Layout Managers and Containers	249
13.1 Layout basics	251
Adding and manipulating child components	251
Size and space negotiation	252
13.2 Box layouts	254
13.3 Grid layouts	255
13.4 Form layouts	257
13.5 Frames	258
13.6 Separators	258
13.7 Notebooks	258
13.8 Scroll areas	261
13.9 Paned windows	262
14 Qt: Widgets	263
14.1 Dialogs	263
Message dialogs	263
Input dialogs	266
Button boxes	267
Custom dialogs	268
Wizards	270
File- and directory-choosing dialogs	270
Other choosers	272
14.2 Labels	272
14.3 Buttons	272
Icons and pixmaps	273
14.4 Checkboxes	274
Groups of checkboxes	274
14.5 Radio groups	276
14.6 Combo boxes	277
14.7 Sliders and spin boxes	279
Sliders	279
Spin boxes	280
14.8 Single-line text	281
Completion	282
Masks and validation	282
14.9 QWebView widget	286
14.10 Embedding R graphics	288
14.11 Drag-and-drop	288
Initiating a drag	289
Handling a drop	289
15 Qt: Widgets Using Data Models	293
15.1 Displaying tabular data	293
Displaying an R data frame	293

Memory management	295
Formatting cells	296
Column sizing	296
15.2 Displaying lists	298
15.3 Model-based combo boxes	299
15.4 Accessing item models	299
15.5 Item selection	300
Accessing the selection	301
Responding to selection changes	302
Assigning the selection	302
15.6 Sorting and filtering	303
15.7 Decorating items	304
15.8 Displaying hierarchical data	307
15.9 User editing of data models	311
15.10 Drag-and-drop in item views	312
15.11 Widgets with internal models	318
Displaying short, simple lists	318
15.12 Implementing custom models	321
15.13 Implementing custom views	325
15.14 Viewing and editing text documents	329
16 Qt: Application Windows	335
16.1 Actions	336
16.2 Menu bars	338
16.3 Context menus	339
16.4 Toolbars	340
16.5 Status bars	341
16.6 Dockable widgets	342
IV The tcltk Package	343
17 Tcl/Tk: Overview	345
17.1 A first example	346
17.2 Interacting with Tcl	347
17.3 Constructors	350
The tkwidget function	352
Geometry managers	352
Tcl variables	353
Commands	354
Themes	354
Window properties and state: tkwininfo	356
Colors and fonts	357
Images	359

17.4	Events and callbacks	360
	The tag	361
	Events	361
	Callbacks	363
	Percent substitutions	364
18	Tcl/Tk: Layout and Containers	369
18.1	Top-level windows	369
18.2	Frames	372
	Label frames	372
18.3	Geometry managers	372
	Pack	373
	Grid	380
18.4	Other containers	385
	Paned windows	385
	Notebooks	386
19	Tcl/Tk: Dialogs and Widgets	389
19.1	Dialogs	389
	Modal dialogs	389
	File and directory selection	390
	Choosing a color	391
19.2	Selection widgets	392
	Check buttons	392
	Radio buttons	394
	Entry widgets	395
	Combo boxes	400
	Scale widgets	402
	Spin boxes	404
20	Tcl/Tk: Text, Tree, and Canvas Widgets	409
20.1	Scroll bars	409
20.2	Multiline text widgets	410
20.3	Menus	415
20.4	Treeview widget	420
	Rectangular data	420
	Editable tables of data	436
	Hierarchical data	436
20.5	Canvas widget	440
	Concept index	447
	Class and method index	449

About this book

Two common types of user interfaces in statistical computing are the command line interface (CLI) and the graphical user interface (GUI). The usual CLI consists of a textual console in which the user types a sequence of commands at a prompt, and the output of the commands is printed to the console as text. The R console is an example of a CLI. A GUI is the primary means of interacting with desktop environments, such as Windows and Mac OS X, and statistical software, such as JMP. GUIs are contained within windows, and resources, such as documents, are represented by graphical icons. User controls are packed into hierarchical drop-down menus, buttons, sliders, etc. The user manipulates the windows, icons, and menus with a pointer device, such as a mouse.

The R language, like its predecessor S, is designed for interactive use through a command line interface (CLI), and the CLI remains the primary interface to R. However, the graphical user interface (GUI) has emerged as an effective alternative, depending on the specific task and the target audience. With respect to GUIs, we see R users falling into three main target audiences: those who are familiar with programming R, those who are still learning how to program, and those who have no interest in programming.

On some platforms, such as Windows and Mac OS X, R has graphical front-ends that provide a CLI through a text console control. Similar examples include the multi-platform RStudioTM IDE, the Java-based JGR, and the RKWard GUI for the Linux KDE desktop. Although these interfaces are GUIs, they are still very much in essence CLIs, in that the primary mode of interacting with R is the same. Thus, these GUIs appeal mostly to those who are comfortable with R programming.

A separate set of GUIs targets the second group of users, those learning the R language. Since this group includes many students, these GUIs are often designed to teach general statistical concepts in addition to R. A CLI component is usually present in the interface, though it is deemphasized by the surrounding GUI, which is analogous to a set of training wheels

on a bicycle. An example of such a GUI is R Commander, which provides a menu- and dialog-driven interface for a wide range of R's functionality and plugin support to extend the functionality.

The third group of users, those who require R only for certain tasks and do not wish to learn the language, are targeted by task-specific GUIs. These interfaces do not usually contain a command line, as the limited scope of the task does not require it. If a task-specific GUI fits a task particularly well, it may even appeal to an experienced user. There are many examples of task-specific GUIs in R. Many GUIs assist in exploratory data analysis, including `exploreRase`, `limmaGUI`, `playwith`, `latticist`, and `Rattle`. Other GUIs are aimed at teaching statistics, e.g., `teachingDemos` and `mosaicManip`. There are a few tools to automatically generate a GUI that invokes a particular R function, such as the `fgui` package and the `guiDlgFunction` function from the `svDialogs` package.

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 or 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. One example is programming a Java/Swing GUI through `rJava`, a native interface between R and Java. It is also possible to extend the `RKward` GUI using a mixture of XML and Javascript, and the `biocep` GUI supports Java extensions. Our focus is instead on programming GUIs with the R language.

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 it 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 `gWidgets`, as it is the most general.

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 to 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 interfaces with the Tk libraries. Although not as modern as GTK+ or Qt, these libraries come pre-installed with the Windows binary, thus bypassing any installation issues for the average end-user. The bindings to Tk were the first ones to appear for R and most of the GUI projects above, notably `Rcmdr`, use this toolkit.

These four main parts are preceded by an introductory chapter on GUIs.

This text is written with the belief that much can be learned by studying examples. There are examples woven through the primary text, as well as stand-alone demonstrations of simple yet reasonably complete applications. The scope of this text is limited to features that are of most interest to statisticians aiming to provide a practical interface to functionality implemented in R. Thus, not every dusty corner of the toolkits will be covered. For the `tcltk`, `RGtk2`, and `qtbase` packages, the underlying toolkits have well documented APIs.

The package `ProgGUIinR` accompanies this text. It 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 `browseWidgetsFiles`, `browseRGtk2Files`, `browseQtFiles`, and `browseTclTkFiles` for browsing the examples from the respective chapters.

The authors would like to thank the following people for their helpful comments made regarding draft versions of this book: Richie Cotton, Erich Neuwirth, Jason Crowley, and Tengfei Yin.

This page intentionally left blank

The Fundamentals of Graphical User Interfaces

1.1 A simple GUI in R

We begin with an example showing how we can use R's standard graphics device as a canvas for a "game" of tic-tac-toe against the computer (Figure 1.1). Although this example has nothing to do with statistics, it illustrates, in a familiar way, some of the issues involved in developing GUIs in R.

Generally, GUIs provide the means for viewing and controlling some underlying data structure. In this example, the data consists simply of information holding the state of the game, defined here in a global variable `board`.

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

A GUI contains one or more views, each of which displays the data in a particular manner. In our case, the view is the game board that we display through an R graphics device. The `layout_board` function creates a canvas for this view:

```
layout_board <- 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:")  
}
```

This example uses a single view; more complex GUIs will contain multiple coordinated, interactive views. The layout of the GUI should help the user navigate the interface and is an important factor in usability. In this case, we benefit from the universal familiarity with the board game.

The user typically sends input to a GUI via a mouse or keyboard. The underlying toolkit allows the programmer to assign functions to be called when some specific event occurs, such as user interaction. Typically, the toolkit *signals* that some action has occurred and then invokes *callbacks* or *event handlers* that have been assigned by the programmer. Each toolkit

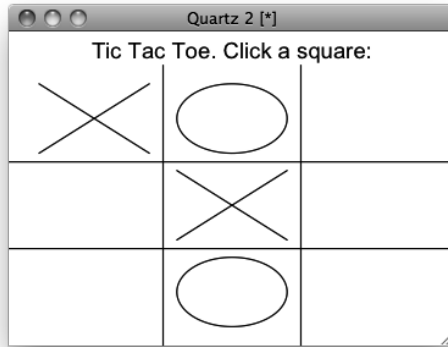


Figure 1.1: Using a graphics device for a game of tic-tac-toe.

has a different implementation. For our game, we will use the locator function built into the base R graphics system:

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

The locator function responds to mouse clicks. We specify how many mouse clicks to gather and the *control* of the program is suspended until the user clicks the sufficient number of times (or somehow interrupts the loop). A GUI that enters a mode in which the flow of a program is blocked and waiting for user input is known as a *modal* GUI. This design is common for simple dialogs that require immediate user attention, although in general a GUI will listen asynchronously for user input.

In the above function `do_play`, `click_handler` is an *event handler*. Its job is to process the output of the locator function, checking first to see 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.

```
click_handler <- function(iloc) {  
  if(is.null(iloc))  
    stop("Game terminated early")  
  move <- floor(unlist(iloc))  
  draw_move(move, "x")  
  board[3*(move[2]-1) + move[1]] <- 1  
  if(!is_finished())  
    do_computer_move()  
  if(!is_finished()) {
```

```

do_play()
}
}

```

The use of `<<-` in the handler illustrates a typical issue in GUI design in R. User input changes the state of the application through callback functions. These callbacks need to modify variables in some shared scope, which may be application-wide or specific to a component. The lexical scoping rules of R, i.e., nesting of closures, has proven to be a useful strategy for managing GUI state. In the above case, we simply modify the global environment, which encloses `click_handler`. When this is inconvenient, direct manipulation of environment objects is sometimes a feasible option. If the scale of the GUI demands more formal mechanisms, we recommend the reference class framework from the `methods` package.

Validation of user input is an important task for a GUI. In the above example, the `click_handler` function checks to see if the user terminated the game early and issues a message.

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 next 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.

```

draw_move <- 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))
  }
}

```

We could use `text` to place a text “x” or “o,” but this may not scale well if the GUI is resized. Most GUI layouts allow for dynamic resizing. This is necessary to handle the variety of data a GUI will display. Even the labels, which one generally considers static, will display different text depending on the language (as long as translations are available).

This function is used to test whether a game is finished:

```

is_finished <- function() {
  (any(abs(rowSums(board)) == 3) ||
   any(abs(colSums(board)) == 3) ||
   abs(sum(diag(board))) == 3 ||
   abs(sum(diag(apply(board, 2, rev)))) == 3)
}

```

```
}
```

The matrix `m` allows us to check easily all the possible ways to get three in a row.

This function picks a move for the computer:

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

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 leftover positions; we leave implementing a better strategy for the interested reader.

Finally, we implement the main entry point for our GUI:

```
play_game <- function() {
  board <- matrix(rep(0,9), nrow=3)
  layout_board()
  do_play()
  mtext("All done\n",1)
}
```

When the game is launched, we first lay out the board and then call `do_play`. When `do_play` returns, a message is written on the screen.

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.

For many GUIs there is a necessary trade-off between usability and complexity. As with any software, there is always the temptation to add features continually without regard for the long-term cost. In this case, there are many obvious improvements: implementing a better artificial intelligence, drawing a line connecting three in a row when there is a win, indicating who won, etc. Adding a feature increases the functionality, at the cost of increased complexity and burden on the user.

1.2 GUI design principles

The most prevalent pattern of user interface design is denoted WIMP, which stands for Window, Icon, Menu, and Pointer. 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 menu bar 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 the 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, in which 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 menu bar and toolbar.

The parameters of an action call, if any, are controlled in sub-windows. These sub-windows are termed *application windows* by Apple^[8], 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 1.2 and 1.3 show three dialogs, representing typical dialogs from the three main operating systems, that perform the same task – collecting 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 various widgets are employed. For example, the control to indicate the number of copies for the Mac is a simple text-entry window, whereas for the KDE and Windows dialog it is a spin button. The latter provides a bit more functionality, for a bit more complexity. The KDE and Mac dialogs have icons to represent actions compactly, whereas the Windows example has none. The landscape icon for the Mac is very clear and provides this feature without having to use a sub-dialog.

[8] Apple Inc. <http://developer.apple.com/>.

1. The Fundamentals of Graphical User Interfaces

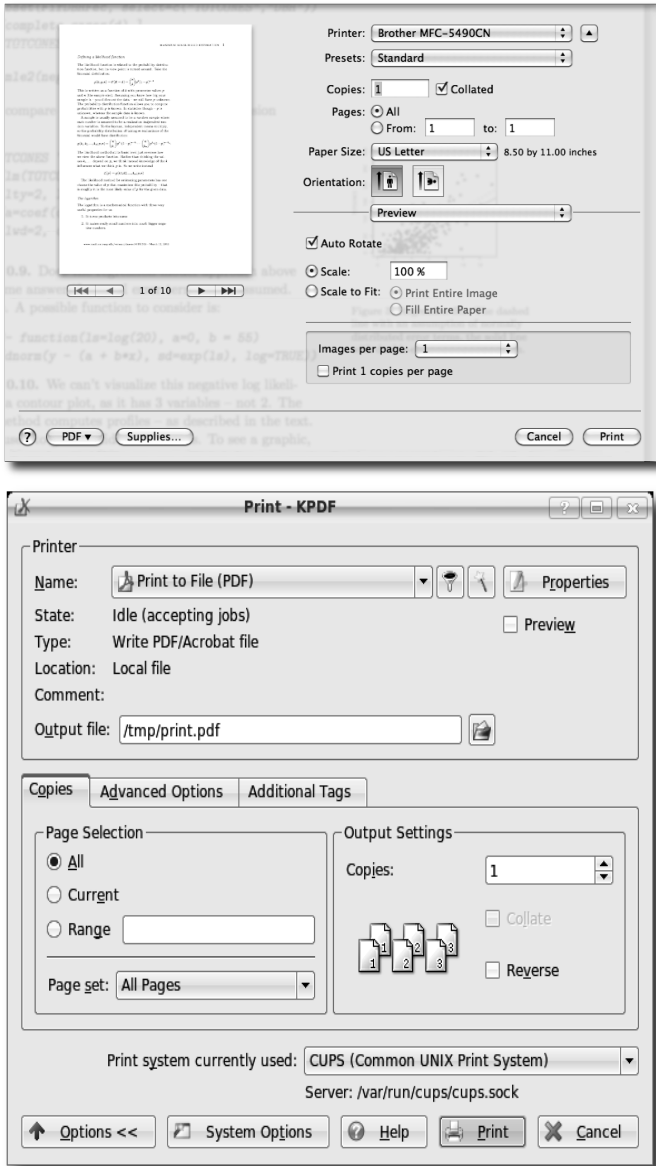


Figure 1.2: Two print dialogs. One from Mac OS X 10.6 and one from KDE 3.5.

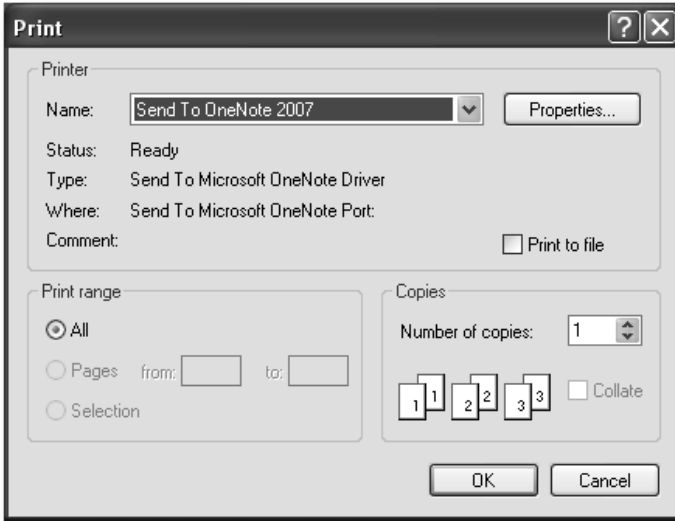


Figure 1.3: R's print dialog under Windows XP using XP's native dialog.

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

Table 1.1: Table of possible selection widgets by data type, size, and selection mode (single or multiple).

Type of data	Single	Multiple
Boolean	checkbox, toggle button	-
Small list	radio button group combo box list box	checkbox group list box
Moderate list	combo box list box	list box
Large list	list box, auto complete	list box
Sequential	slider spin button	
Tabular	table	table
Hierarchical	tree	tree

The Mac GUI puts the number of pages in focus, whereas Windows places the printer in focus. Focus allows the user to interact with the GUI without the mouse. Typically, the tab key is used to step through the controls. GUIs often have shortcuts that allow power users to initiate actions or shift the focus directly to a specific widget through the keyboard. 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 they cannot be seen in the figure. The use of conventional icons on the buttons also helps guide the user.

1.3 Controls

This section provides an overview of many common controls, i.e., widgets that accept input, display data, or provide visual guides to help the user navigate the interface. If the reader is already familiar with the conventional types of widgets and how they are arranged on the screen, this section and the next should be considered optional.

Choice of control

A GUI comprises one or more widgets. The appropriate choice 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 the space available in the layout. As an example, Table 1.3 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 1.4 shows several such controls in a single dialog. A checkbox enables an intercept, a radio group selects either full factorial or a custom model, a combo box 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 1.4 shows an SPSS dialog for specifying 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 interface that balances the needs of both types of user, or the SPSS interface may be appropriate. Knowing the potential user base is important.

Presenting options

The widgets that receive user input need to translate that input into a command that modifies the state of the application. Commands, like R functions, often have parameters, or options. For many options, there is a discrete set of possible choices, and the user needs to select one of them. Examples include 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.

Checkboxes

A *checkbox* specifies a value for a logical (Boolean) option. 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.

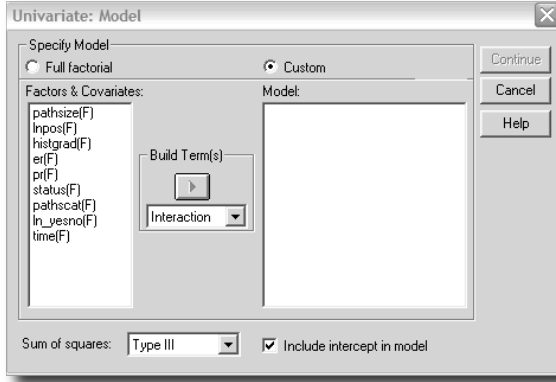


Figure 1.4: A dialog box from SPSS version 11 for specifying terms for a linear model. The graphic shows a dialog that allows the user to specify individual terms in the model using several types of widgets for selection of values, such as a radio button group, a checkbox, combo boxes, and list boxes.

Radio buttons

A *radio button group* selects exactly one value from a vector of possible values. The analogy dates back to old car radios where there were a handful of buttons for selecting preset channels. When a new button was pushed in, the previously pressed button popped out. 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 fill the available space. Figure 1.5 uses radio button groups for choosing the distribution, kernel and sample size for the density plot.

Combo boxes

A *combo box* is similar to a radio button group, in that it is used to select one value from several. However, a combo box displays only the value currently selected, which reduces visual complexity and saves space, at the cost of an extra click to show the choices. Toolkits often combine a combo box with a text entry area for specifying an arbitrary value, possibly one that is not represented in the set of choices. A combo box is generally desirable over radio buttons when there are more than four or five choices. However, the combo box also has its limits. For example, some web forms require choosing a country from a list of hundreds. In such cases, features such as incremental type-ahead search are useful.

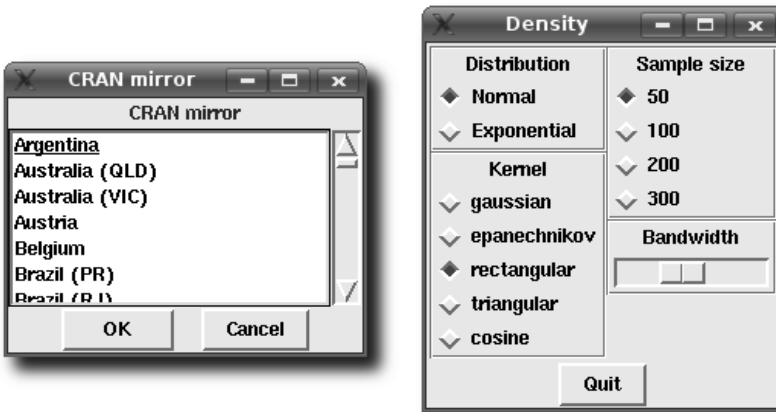


Figure 1.5: Two applications of the `tcltk` package. The left graphic is produced by `chooseCRANmirror` and uses a list box to allow selection from a long list of possibilities. The right graphic is the `tkdensity` demo from the `tcltk` package. It uses radio buttons and a slider to select the parameter values for a density plot.

List boxes

A *list box* displays a list of possible choices in a column. While the radio button group and combo box select only a single value, a list box supports multiple selection. Another difference is that the number of displayed choices depends dynamically on the available space. If a list box contains too many items to display simultaneously, a scroll bar is typically provided for adjusting the visible range. Unlike with the combo box, the choices are immediately visible to the user. Figure 1.5 shows a list box created by the R function `chooseCRANmirror`. There are too many mirrors to fit on the screen, but a combo box would not take advantage of the available space. The list box is a reasonable compromise.

Sliders and spin buttons

A *slider* is a widget that selects a value from a sequence of possible values, typically through the manipulation of a knob that moves or “slides” along a line that represents the range of possible values. Some toolkits generalize beyond a numeric sequence. The slider is a good choice for offering the user a selection of ordinal or numerical parameter values. For example, the letters of the alphabet could be a sequence. The `tkdensity` demo of the `tcltk` package (Figure 1.5) uses a slider to adjust the bandwidth of a density estimate dynamically.

A *spin button* plays a similar role to the slider, in that it selects a value within a set of bounds. Typically, this widget is drawn with a text box displaying the current value and two arrows to increment or decrement the selection. Usually, the text box can be edited directly. A spin button has the advantage of using less screen space, and directly entering a specific value, if known, is easier than selecting it with a slider. One disadvantage is that the position of the selected value within the range is not as obvious as with the slider. As a compromise, combining a text box with a slider is possible and often effective. A spin button is used in the KDE print dialog of Figure 1.2 to adjust the number of copies.

Initiating an action

After the user has specified the parameters of an action, typically by interacting with the selection widgets presented above, it comes time to execute the action. Widgets that execute actions include the familiar buttons, which are often organized into menu bars and toolbars.

Buttons

A *button* issues commands when invoked, usually via a mouse click. In Figure 1.2, the “Properties” button, when clicked, opens a dialog for setting printer properties. The button with the wizard icon also opens a dialog. As buttons execute an action, they are often labeled with a verb.^[8] In Figure 1.4 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. Actions may be given shortcut bindings, and their button proxies typically reflect the proper key combination to invoke the action. The KDE print dialog in Figure 1.2 has these bindings indicated via the underlined letter on the button labels.

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, in which an icon is used to complement or replace a text label on a button or other control. A button represents an action, so an icon on a button should visually depict an action.

Menu bars

Menus play a central role in the WIMP desktop. The *menu bar* contains items for many of the actions supported by the application. By convention, menu bars are associated with a top-level window. This is enforced by some toolkits and operating systems but not all. In Mac OS X, the menu bar appears on the top line of the display, but other platforms place the menu bar at the top of the top-level window. In a statistics application, the “document” may be the active data frame, a report, or a graphic.

The styles used for menu bars are fairly standardized, as this allows new users to orient themselves quickly within a GUI. The visible menu names are often in the order *File*, *Edit*, *View*, *Tools*, 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 conventionally uses a “...” to indicate that a subsequent dialog will open so that more information can be gathered to complete the action. The text may also indicate a keyboard accelerator, such as *Find Next F3*, indicating that both “N” as a keyboard accelerator and F3 as a shortcut will initiate this same action. (Shortcuts are not translated, but keyboard accelerators must be. As such, they are less frequently used. In particular, keyboard accelerators are not supported in Mac OS X menus.)

Not all actions will be applicable at any given time. It is recommended, that rather than deleting these menu items, disable them (grayed out) instead.

Menus may come to contain many items. To help the user navigate, menu items are usually grouped with either horizontal separators or hierarchical submenus.

The use of menus has evolved to allow the user to view and control properties of the application state. There may be checkboxes drawn next to the menu item or an icon indicating the current state.

Another use of menus is to bind contextual menus (pop-up menus) to certain mouse clicks on GUI elements. Typically, a right mouse click 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 menu bar. 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.

Action objects

When clicking on a button, the user expects some “action” to occur. For example, a save dialog is summoned, or a page is printed. GUI toolkits commonly represent such actions as formal, invisible objects that are proxied by widgets, usually buttons, on the screen. Often, all of the primary commands supported by an application have a corresponding action object, and the buttons associated with those actions are organized into menu bars and toolbars.

An action object is essentially a data model, with each proxy widget acting as a view. Common components of an action include a textual label, an icon, perhaps a shortcut, and a handler to call when the action is selected.

Modal dialogs

A *modal dialog box* is a dialog box that keeps the focus until the user takes an action to dismiss the box. It prompts a user for immediate input, such as asking for confirmation when overwriting a file. Modal dialog boxes can be disruptive to the flow of interaction, so they are used sparingly. As the control flow is blocked until the window is dismissed, functions that display modal dialogs can return a value when an event occurs, rather than have a handler respond to asynchronous input. The `file.choose` function, mentioned below, is a good example. When this function is called during an interactive R session, the user is unable to interact with the command line until a file has been specified or the dialog dismissed.

Message dialogs

A *message dialog* is a high-level dialog widget for communicating a message to the user. By convention, 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, often labeled “Ok,” to dismiss the dialog. Additional buttons/responses are possible. The *confirmation dialog* variant would add a “Cancel” button, which would invalidate the proposed action.

File choosers

A file chooser allows for the selection of files and directories. 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.

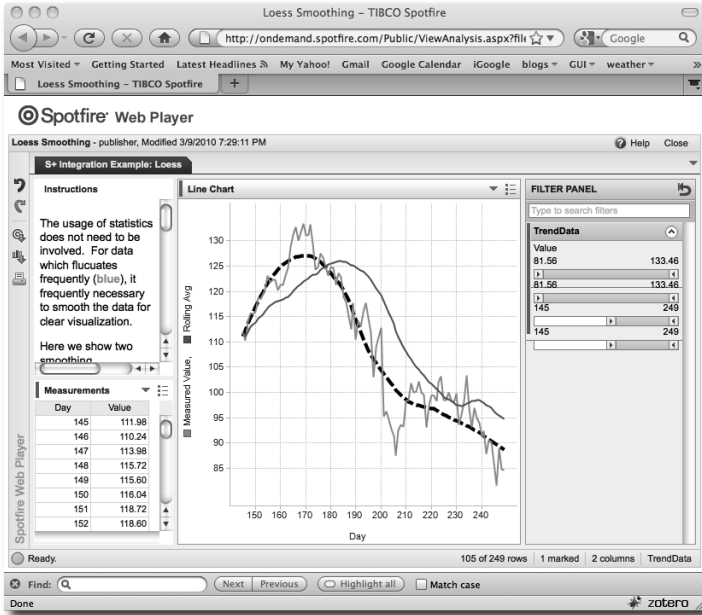


Figure 1.6: This screen shot from Tibco’s Spotfire web player illustrates a table widget (lower left) displaying the cases that are summarized in the graphic. The right bar filters the cases in the table.

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. All the toolkits we discuss have some means to embed R’s graphics.

Tabular display

A *table widget* shows tabular data, such as a data frame, in which each column has a specific data type and cell-rendering strategy. Table widgets handle the display, sorting, and selection of records from a dataset. Depending on the configuration of the widget, cells may be editable. Figure 1.6 shows a table widget in a Spotfire web player demonstration.

Tree widgets

So far, we have seen how list boxes display homogeneous vectors of data, and how table widgets display tabular data, such as 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 user session (Figure 1.7). 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.

Displaying and editing text

The letter P in WIMP stands for “pointer,” so it is not surprising that WIMP GUIs are designed around the pointing device. The keyboard is generally relegated to a secondary role, in part because it is difficult to type and move the mouse at the same time. For statistical GUIs, especially when integrating with the command-line interface of R, the flexibility afforded by arbitrary text entry is essential for any moderately complex GUI. Toolkits generally provide separate widgets for text entry depending on whether the editor supports a single line or multiple lines.

Single lines of text

A text-entry widget for editing a single line of text is found in the KDE print dialog (Figure 1.2). It specifies the page range. Specifying a complex page range, which might include gaps, would require a complex point-and-click interface. In order to avoid complicating the GUI for a feature that is rarely useful, a simple language has been developed for specifying page ranges. There is overhead involved in the parsing and validation of such a language, but it is still preferable to the alternative.

Text-editing boxes

Figure 1.8 shows three multiline text entries in an Rcmdr window. It provides an R console and status message area. The “Output Window” demonstrates the utility of formatting attributes. In this case, attributes specify the color of the commands, so that the input can be distinguished from the output.

Guides and feedback

Some widgets display information but do not respond to user input. Their main purpose is to guide the user through the GUI and to display feedback and status messages. Communicating application status, such as during

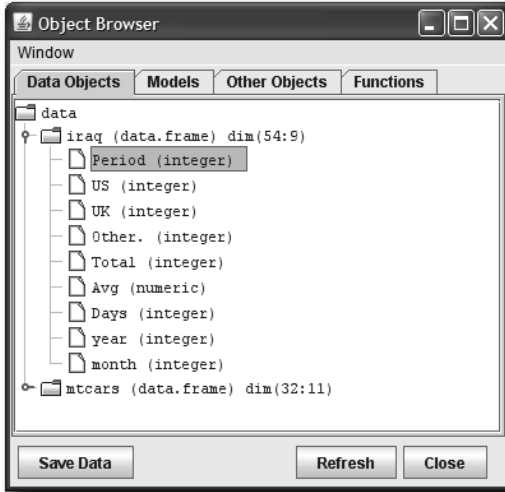


Figure 1.7: The object browser in the JGR GUI uses a tree widget to display the possibly hierarchical nature of R objects.

long-running calculations or when errors occur, is an often over-looked but critically important feature of any effective GUI.

Labels

A label is a widget for placing text into a GUI that is typically not intended for editing, or even for selecting with a mouse. The main role of a label is to describe another component of the GUI. Most toolkits support rich text in labels. Figure 1.8 shows labels marked in red and blue in `tcltk`.

Status bars

A statusbar displays general status messages, as well as feedback on actions initiated by the user, such as progress or errors. Messages replace the previous message and may disappear after a certain period of time. In the traditional document-oriented GUI, statusbars are placed at the bottom.

Related to status bars are info bars or alert boxes, which allow a programmer to display a transient message dialog that emerges from either the top or bottom of the application window. An example is the Firefox dialog that asks whether Firefox should remember a password entered on the previous page. It appears just below the toolbar and disappears automatically as the user continues to browse.

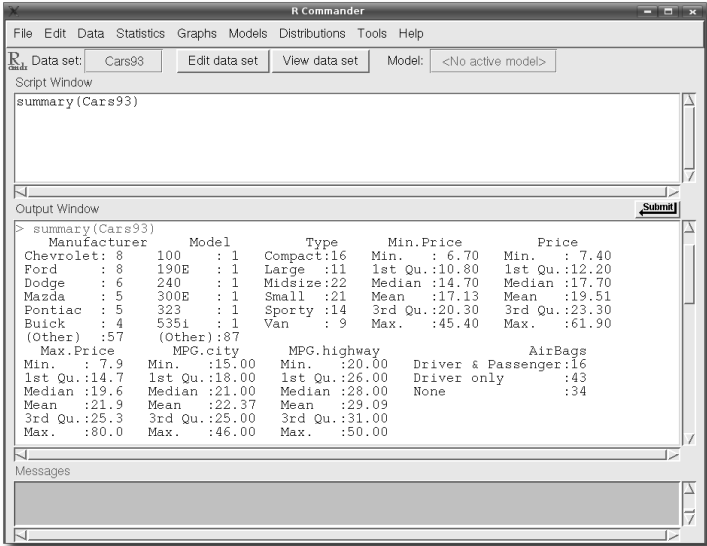


Figure 1.8: Screenshot showing the main Rcmdr (1.3-11) window illustrating the use of multiline text-entry areas for a command area, an output area, and a message area.

Tooltips

A tooltip is a small window that is displayed when a user hovers the mouse over a tooltip-enabled widget. Tooltips are an embellishment 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.

Progress bars

A progress bar indicates progress on a particular task, which may or may not be bounded. A bounded progress bar usually reports progress in terms of percentage completed. Progress bars should be familiar, as they are often displayed during software installation and while downloading a file. For long-running statistical procedures they can give useful feedback that something is happening.

1.4 Containers

The KDE print dialog of Figure 1.2 contains many of the widgets we discussed in the previous section. Before we can create such a dialog, we

need to discuss the positioning of widgets on the screen. This process is called *widget layout*.

A layout emerges from the organization of the widgets into a hierarchy, in which a parent widget positions its children within its allocated space. The top-level window is parentless and forms the root of the hierarchy. A parent visually contains its children and thus is usually called a *container*. This design is natural, because almost every GUI has a hierarchical layout. It is easy to apply a different layout strategy to each region of a GUI, and when a parent is added or removed from the GUI, so are its children.

It is sometimes tempting for novices to assign simply a fixed position and dimensions for every widget in a GUI. However, such static layouts do not scale well to changes in the state of the application or simply changes in the window size dictated by the window manager. Thus, it is strongly encouraged to delegate the responsibility of layout to a *layout manager*, which dynamically calculates the layout as constraints change. Depending on the toolkit, the layout manager might be the container itself, or it might be a separate object to which the container delegates.

Regardless, the type of layout is generally orthogonal to the type of container. For example, a container might draw a border around its children, and this would be independent of how its children are laid out. The rest of this section is divided into two parts: container widgets and layout algorithms. We will continually refer back to the KDE print dialog example as we proceed.

Top-level windows

The top-level window of a GUI is the root of the container hierarchy. All other widgets are contained within it. The conventional main application window will consist of a menubar, a toolbar and a status bar. The primary content of the window is inserted between the toolbar and the status bar, in an area known as the *client area* or *content area*. In the case of a dialog, the content usually appears above a row of buttons, each of which represents a possible response. The print dialog conforms to the dialog convention. The print options fill the content area, and there is a row of buttons at the bottom for issuing a response, such as "Print."

A window is typically decorated with a title and buttons to iconify, maximize, or close. In the case of the print dialog, the top-level window is entitled "Print - KPDF." 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.

Once a window is shown, its dimensions are managed by the user, through the window manager. Thus, the programmer must size the window before it becomes visible. This is often referred to as the "default"

size of the window. Positioning of a top-level window is generally left to the window manager.

The top-level window forwards window-manager events to the application. For example, an application might listen to the window-close event in order to prompt a user if there are any unsaved changes to a document.

Tabbed notebooks

A notebook widget depicts each child as if it were a page in a notebook. A page is selected by clicking on a button that appears as a tab. Only a single child is shown at a time. The tabbed notebook is a space-efficient, categorizing container that is most appropriate when a user is interested in only one page at a time. Modern web browsers take advantage of the tabbed notebook to allow several web pages to be open at once within the same window. In the KDE print dialog, detailed options are collapsed into a notebook in order to save space and organize the many options into simple categories: “Copies,” “Advanced Options,” and “Additional Tags.”

Frames

A frame is a simple container that draws a border, possibly with a label, around its child. The purpose of a frame is to enhance comprehension of a GUI by visually distinguishing one group of components from the others. The displayed page of the notebook in Figure 1.2 contains two frames, visually grouping widgets by their function: either Page Selection or Output Settings.

Expanding boxes

An expanding container, or box, will show or hide its children according to the state of a toggle button. By way of analogy, radio buttons are to notebooks as check buttons are to expanding containers. An expanding box allows the user to adapt a GUI to a particular use case or mode of operation. Often, an expanding box contains so-called “advanced” widgets that are only occasionally useful and are of interest only to a small percentage of the users. For example, the Options button in Figure 1.2 controls an expanding box that contains the print options, which are usually best left to their defaults.

Paned boxes

Usually, a layout manager allocates screen space to widgets, but sometimes the user needs to adapt the allocation. For example, the user may wish to increase the size of an image to see the fine details. The *paned container* supports this by juxtaposing panes, either vertically (stacked) or

horizontally. The area separating the panes, sometimes called a *sash*, can be adjusted by the user with the mouse.

Layout algorithms

Box layout

The box layout is the most common type of layout algorithm for positioning child components. A box will pack its children either horizontally or vertically.¹ Usually, the widgets are packed from left to right, for horizontal boxes, or from top to bottom, in the case of a vertical box. The upper-left figure in Figure 1.9 illustrates these possibilities.

The box layout needs to allocate space to its children in both the vertical and horizontal directions. The typical box layout algorithm begins by satisfying the minimum size requirements of its children. The box may need to request more space for itself in order to meet the requirements.

Once the minimum requirements are satisfied, it is conventional and usually desirable for the widgets to fill the space in the direction orthogonal to the packing. For example, widgets in a horizontal box will fill all of their vertical space (the upper-right graphic in Figure 1.9 shows some fill possibilities). When this is not desired, most box widgets support different ways of vertically (or horizontally) aligning the widgets (the lower-left graphic in Figure 1.9).

More complex logic is involved in the allocation of space in the direction of packing. Any available space after meeting minimum requirements needs to be either allocated to the children or left empty. This depends on whether any children are set to expand. The available space will be distributed evenly to all expanding children. Each child may fill that space or leave it empty. The non-expanding children are simply packed against their side of the container. If there are no expanding children, the remaining space is left empty in the middle (or end, if there are no widgets packed against the other side). See the lower-right panel in Figure 1.9. One could think of this space as being occupied by an invisible spring. Invisible expanding widgets also act as springs.

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

¹With exceptions: the `pack` command of `tcltk` can mix the two directions.

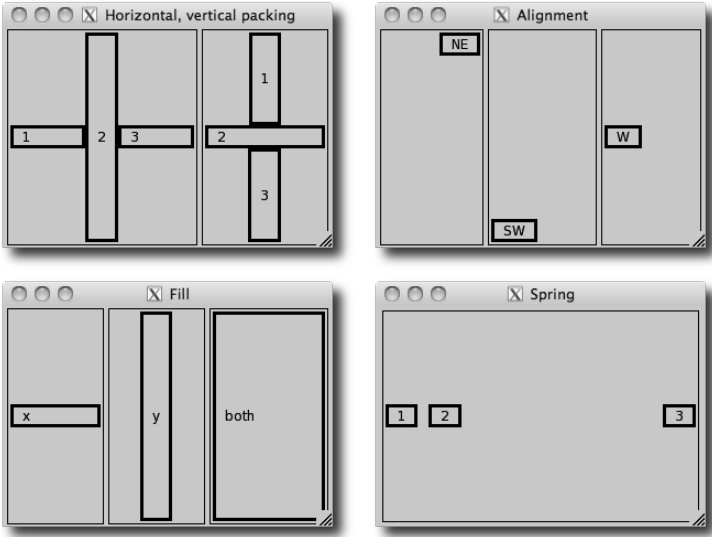


Figure 1.9: Different possibilities for packing child components within a box. The upper left shows horizontal and vertical layout. The upper right shows some possible alignments or anchorings. The lower left shows that a child could “expand” to fill the space either horizontally, vertically, or both. The lower right shows both a fixed amount of space between the children and an expanding spring between the child components.

Grid layout

The box layout algorithm typically aligns its children along a single dimension. The horizontal box, for example, aligns its children vertically. Nevertheless, nesting permits the construction of complex layouts using only simple boxes. It is sometimes desirable to align widgets in both dimensions, i.e., to lay them out on a grid. The most flexible grid layout algorithms allow non-regular sizing of rows and columns, as well as the ability for a widget to span multiple cells. Usually, a widget fills the cells allocated to it, but if this is not possible, it may be anchored at a specific point within its cell.

Part I

The gWidgets Package

This page intentionally left blank

gWidgets: Overview

The `gWidgets` package provides a convenient means to create rapidly small to medium-size GUIs within R. The package provides an abstract interface for the other graphical toolkits discussed in this text, allowing for similar access to each. Unlike the underlying toolkits, `gWidgets` has relatively few constructors and methods. Basically, the entire set is enumerated in Tables 2.2, 3.1, 3.2, and 4.1. This means `gWidgets` is relatively easy to learn, allowing for rapid prototyping. (It also means that as projects progress, one might need to move to a more powerful underlying toolkit.)

Typical uses of GUIs written in R involve teaching demos and, sharing functionality with less technically proficient colleagues, etc. In many cases the end user may have a different operating system or different set of graphical libraries installed. The underlying toolkits supported by `gWidgets` are all cross- platform, and `gWidgets` code is mostly-cross toolkit, although differences do come up. (Compare, for example, the same code realized on different operating systems and toolkits in Figure 2.1.) This means, there is a good chance that code you write can be shared easily with someone else.

The `gWidgets` package started as a port to `RGtk2` of Simon Urbanek's `iWidgets` package which was written for Swing through `rJava`^[12]. Along the way, `gWidgets` was extended and abstracted to work with other GUI toolkit backends available for R. A separate package provides the interface. As of writing there are interfaces for `RGtk2`, `qtbase`, and `tcltk`. The `gWidgetsWWW2` package provides a similar interface for web programming, but there are enough differences that we will not discuss it further.

We jump right in with an example and leave comments about installation to the end of the chapter. The following shows some sample `gWidgets` commands that set up a basic interface allowing a user to search his or her

[12] Simon Urbanek. `iWidgets` - Basic GUI widgets for R. <http://www.rforge.net/iWidgets/index.html>.

2. gWidgets: Overview

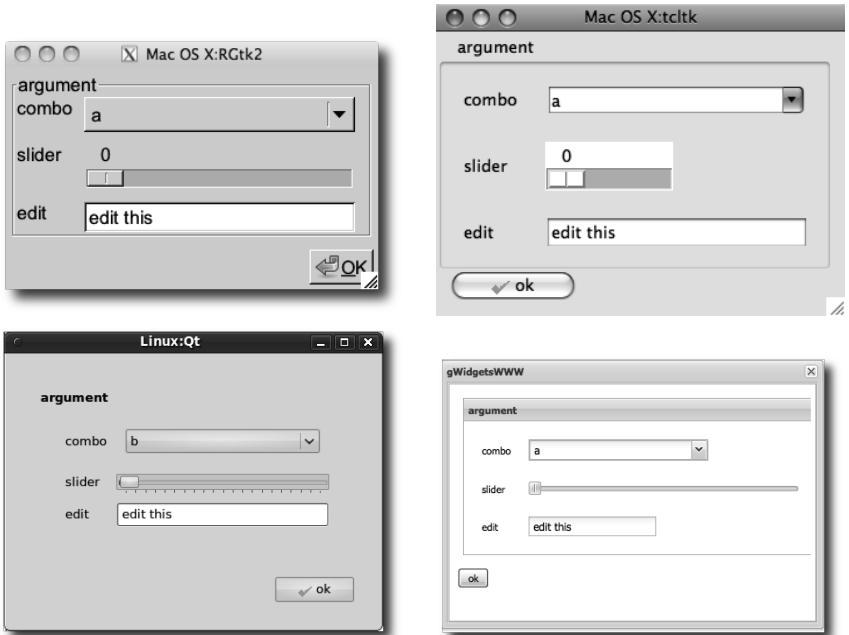


Figure 2.1: The `gWidgets` package works with different operating systems and different GUI toolkits. This shows, the same code using the `RGtk2`, `tcltk`, `qtbases` packages for a toolkit. Additionally, the `gWidgetsWWW` package is used in the lower right figure.

hard drive for files matching a user-specified pattern.¹ The first line loads the package; the others will be described in the following.

```
require(gWidgets)
options(guiToolkit="RGtk2")
##
window <- gwindow("File search", visible=FALSE)
paned <- gpanedgroup(cont = window)
## label and file selection widget
group <- ggroup(cont = paned, horizontal = FALSE)
glabel("Search for (filename):", cont=group, anchor=c(-1,0))
txt_pattern <- gedit("", initial.msg = "Possibly wildcards",
                    cont = group)
##
glabel("Search in:", cont = group, anchor = c(-1,0))
start_dir <- gfilebrowse(text = "Select a directory ...",
                        quote = FALSE,
```

¹Many thanks to Richie Cotton for suggesting this example and its follow-up in Example 4.5.

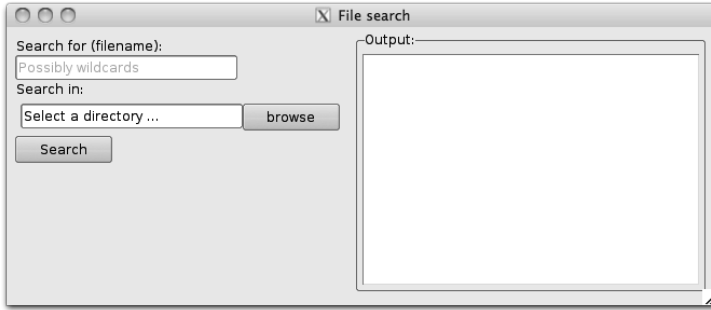


Figure 2.2: A simple GUI for search for files matching a pattern. This GUI uses a paned group to separate the controls for searching from the results.

```

                                type = "selectdir", cont = group)
## A button to initiate the search
search_button <- gbutton("Search", cont = group)
addSpring(group)
## Area for output
frame <- gframe("Output:", cont = paned, horizontal = FALSE)
search_results <- gtext("", cont = frame, expand = TRUE)
size(search_results) <- c(350, 200)
## add interactivity
addHandlerChanged(search_button, handler = function(h,...) {
  pattern <- glob2rx(svalue(txt_pattern))
  file_names <- dir(svalue(start_dir), pattern,
                   recursive = TRUE)
  if(length(file_names))
    svalue(search_results) <- file_names
  else
    galert("No matching files found", parent = window)
})
## display GUI
visible(window) <- TRUE

```

This example shows several different widgets being used to construct a GUI, as seen in Figure 2.2. For example, on the left is a text-entry widget (`gedit`), a directory-browsing widget (`gfilebrowse`) and a button (`gbutton`). On the right is a multiline text widget (`gtext`) in a framed container (`gframe`).

2.1 Constructors

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

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

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

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

The handler and action arguments The handler and action arguments are used to pass in event handlers. We discuss those in Section 2.3.

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

The ... argument Not only is the `...` mechanism used to pass in arguments to the `add` method of the parent container, it may also used to pass in additional values to the constructor in the toolkit package. Some toolkits allow additional functionality beyond that specified in the basic API. Any documentation for these additional arguments appears in the toolkit package.

The toolkit argument The toolkit argument is usually not specified. It is there to allow the user to mix toolkits within the same R session, but in practice this can cause problems due to competing event loops. The default for the `toolkit` argument is to call `guiToolkit`. This function will check whether a toolkit has been specified, or if only one is available. If neither case is so, then a menu will be provided for the user to choose one. In our example we have called

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

to explicitly set the toolkit.

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

- Containers: such as the top-level window `window`, the paned group `paned`, or the frame `frame` (Table 3.1)
- Components: such as the unnamed labels, the edit area `txt_pattern`, or the button `search_button` (Tables 4.1 and 5.1)
- Dialogs: such as `galert` and `gfilebrowse` (Table 2.4)

2.2 Methods

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

GUI objects have a state determined by one or more of their properties. In `gWidgets`, many properties are set at the time of construction. However, there are also several methods to adjust these properties for `gWidgets` objects.²

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

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

```
svalue(txt_pattern)
```

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

```
svalue(search_results) <- file_names
```

to update the text for the multi-line text widget `search_results`.

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

For these selection widgets the familiar `[]` and `[<-` methods refer to the underlying data to be selected from.

² We are a bit imprecise about the term “method” here. The `gWidgets` methods call further methods in the underlying toolkit interface, which we think of as a single method call. The actual S4 object has a slot for the toolkit and the widget created by the toolkit interface to dispatch on.

The call

```
visible(window) <- TRUE
```

sets the visibility property of the top-level window. In our example, the `gwindow` constructor is passed `visible=FALSE` to suppress an initial drawing, making this call to `visible<-` necessary to show the GUI. The `visible<-` generic has different interpretations for the various widgets.

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 whether a widget is sensitive to user input.

The underlying toolkit widget The `gWidgets` API provides just a handful of generic functions for manipulating an object's properties compared to the number of methods typically provided by a GUI toolkit for a similar object. Although this simplicity makes `gWidgets` easier to work with, one may wish to access the underlying toolkit object to take advantage of a richer API. In most cases, the `getToolkitWidget` will provide that object. For convenience, the method `$` is implemented to call a method on the underlying toolkit widget, and the methods `[[` and `[[<-` are implemented to inspect and set properties of the underlying widget. We will not illustrate here though, as we try to stay toolkit-agnostic in our examples.

2.3 Event handlers

In our example, the search button is created with:

```
search_button <- gbutton("Search", cont = group)
```

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

```
addHandlerChanged(search_button, handler = function(h,...) {
  pattern <- glob2rx(svalue(txt_pattern))
  file_names <- dir(svalue(start_dir), pattern, recursive=TRUE)
  if(length(file_names))
    svalue(search_results) <- file_names
  else
    galert("No matching files found", parent = window)
})
```

Callbacks in `gWidgets` have a common signature `(h,...)`. The first argument is a list with components `obj`, to pass in the receiver of the event

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

Method	Description
<code>svalue</code> , <code>svalue<-</code>	Get or set widget's main property
<code>size<-</code>	Set preferred size request of widget in pixels
<code>show</code>	Show widget if not visible
<code>dispose</code>	Destroy widget or its parent
<code>enabled</code> , <code>enabled<-</code>	Adjust sensitivity to user input
<code>visible</code> , <code>visible<-</code>	Show or hide object or part of object
<code>focus<-</code>	Set focus to widget
<code>insert</code>	Insert text into a multiline text widget
<code>font<-</code>	Set a widget's font
<code>update</code>	Update widget value
<code>isExtant</code>	Does R object refer to GUI object that still exists
<code>[</code> , <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>getToolkitWidget</code>	Return underlying toolkit widget for low-level use

(the button in this case), and action to pass along any value specified by the action argument.

For example, a typical idiom within a callback is

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

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

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

The package provides a number of generic methods (Table 2.3) to add callbacks for different events beyond `addHandlerChanged`, which is used to assign a callback for the typical event for the widget, such as the clicking of a button. We refer to these methods with "addHandlerXXX," where the

XXX describes the event. These are useful when more than one event on that widget is of interest. For example, for single-line text widgets, such as `txt_pattern` in our example, the `addHandlerChanged` method sets a callback to respond when the user finishes editing, whereas a handler set by `addHandlerKeystroke` is called each time a key is pressed.

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

```
search_button <- gbutton("Search", cont = group,
  handler = function(h,...) {
    pattern <- glob2rx(svalue(h$action$txt))
    file_names <- dir(svalue(h$action$dir),
      pattern, recursive = TRUE)
    if(length(file_names))
      svalue(h$action$results) <- file_names
    else
      galert("No matching files found", parent = w)
  },
  action = list(txt = txt_pattern, dir = start_dir,
    results = search_results)
)
```

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

The `addHandlerXXX` methods return an ID. This ID can be used with the method `removeHandler` to remove the callback, or with the methods `blockHandler` and `unblockHandler` to block temporarily 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 for quickly making some basic dialogs for showing messages or gathering information. Mostly these are modal dialogs that take control of the event loop, not allowing any other part of the GUI to be active for interaction. As such, in `gWidgets`, constructors of modal dialogs do not return an object to manipulate through its methods, but rather return the user response to the dialog. For example, the `gfile` dialog, described later, is a modal dialog that pops up a means to select a file returning the selected file path or NA. It is used along the lines of:

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

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>	Set handler for when widget is clicked with (left) mouse button. May return position of click through components <code>x</code> and <code>y</code> of the <code>h-list</code> .
<code>addHandlerDoubleClick</code>	Set handler for when widget is double-clicked.
<code>addHandlerRightclick</code>	Set handler for when widget is right-clicked.
<code>addHandlerKeystroke</code>	Set handler for when key is pressed. The key component is set to this value, if possible.
<code>addHandlerFocus</code>	Set handler for when widget gets focus.
<code>addHandlerBlur</code>	Set handler for when widget loses focus.
<code>addHandlerExpose</code>	Set handler for when widget is first drawn.
<code>addHandlerUnrealize</code>	Set handler for when widget is undrawn on screen.
<code>addHandlerDestroy</code>	Set handler for when widget is destroyed.
<code>addHandlerMouseMotion</code>	Set handler for when widget has mouse go over it.
<code>addDropSource</code>	Specify a widget as a drop source.
<code>addDropMotion</code>	Set handler to be called when an item is dragged over the widget.
<code>addDropTarget</code>	Set handler to be called on a drop event. Adds the component <code>dropdata</code> .
<code>addHandler</code>	(Not cross-toolkit) Allows one to specify an underlying signal from the graphical toolkit and handler.
<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 pop-up menu to widget.
<code>add3rdMousePopupMenu</code>	Bind popup menu to right mouse click.

Table 2.3: Table of constructors for basic dialogs in gWidgets.

Constructor	Description
<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.
<code>gfile</code>	File and directory selection dialog.

In the example, we use two non-modal dialogs: `gfilebrowse` to select a directory; and `galert` to display a transient message, if no files are found through our search. Here, we describe the dialogs that can be used to display a message or gather a simple amount of text. The `gfile` dialog is described in Section 4.4, and `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 pop up that allows the user to select a value which will be returned as `TRUE` or `FALSE`:

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

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

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

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

For example,

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

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

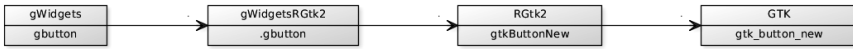


Figure 2.3: The construction of a button widget in `gWidgets` requires several steps

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.

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

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

This illustrates how to use the return value.

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

2.5 Installation

The `gWidgets` package interfaces with an underlying R package through an intermediate package. Figure 2.3 shows the sequence of calls to produce a button. First the `gWidgets` package dispatches to a toolkit package (`gWidgetsRGtk2`), which in turn calls functions in the underlying R package (`RGtk2`) which in turn calls into the graphical toolkit to produce an object. This is then packaged into an S4 object to manipulate.³

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

The difficulty for the end user is the installation of the graphic toolkit, as all other packages are installed through CRAN or are recommended

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

Table 2.4: Installation notes for GUI toolkits.

	Gtk+	Qt	Tk
Windows	Installed by RGtk2	Included with qtbase	In binary install c
Linux	Standard	Standard	Standard
OS X	Download binary .pkg	Vendor supplied	In binary install c

packages with an R installation (`tcltk`). Table 2.4 roughly describes the installation process for different operating systems and toolkits. For Windows users, some details are linked to in the R for Windows FAQ.

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

gWidgets: Container Widgets

After identifying the underlying data to manipulate and deciding how to represent it, GUI construction involves three basic steps:

- creation and configuration of the main components,
- the layout of these components, and
- connecting the components through callbacks to make a GUI interactive.

This chapter discusses the layout process within `gWidgets`. Layout in `gWidgets` is done by placing child components within parent containers, which in turn may be nested in other containers.¹ In our file-search example from the previous chapter, we nested a framed box container inside a paned container inside a top-level window.

The `gWidgets` package provides just a few types of containers: top-level windows (`gwindow`), box containers (`ggroup`, `gframe`, `gexpandgroup`), a grid container (`glayout`), a paned container (`gpanedgroup`), and a notebook container (`gnotebook`). Figure 3.1 shows most of these employed to produce a GUI to select and then show the contents of a file.

In some toolkits, notably `tk`, the widget constructors require the specification of a parent container for the widget. To accommodate that, the `gWidgets` constructors – except for top-level windows and dialogs – have the argument `container` to specify the immediate parent. Within the constructor is the call `add(container, child, ...)`, where the constructor creates the child and `...` values are passed from the constructor down to the `add` method. That is, the widget construction and layout are coupled together. Although this isn't necessary when utilizing `RGtk2` or `qtbase` – and the two aspects can be separated – for the sake of cross-toolkit portability we do not illustrate this style here.

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

3. gWidgets: Container Widgets

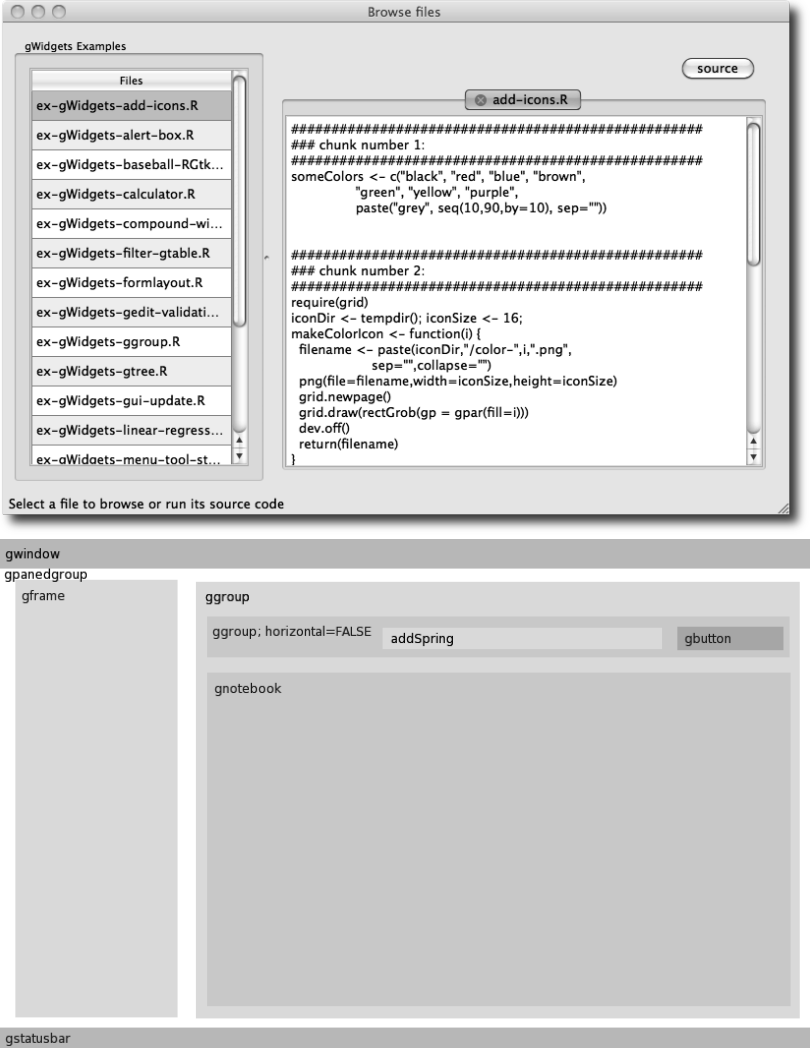


Figure 3.1: The example browser for gWidgets showing different layout components. The lower image shows the containers used.

3.1 Top-level windows

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

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

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

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

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

The optional `visible` argument, used above with its default value `TRUE`², controls whether the window is initially drawn. If it's not drawn, the `visible<-` method, taking a logical value, can be used to draw the window later. Often it is good practice to suppress the initial drawing, especially for displaying GUIs with several controls, as the incremental drawing of subsequent child components can make the GUI seem sluggish. As well, this allows the underlying toolkit to compute the necessary size before it is displayed.³

For example, a typical usage follows this pattern:

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

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

The initial placement of a window " $x + y$ " will be decided by the window manager, unless the parent argument is specified. If this is done with a vector of x and y pixel values, the upper-left corner will be placed at this point. The parent argument can also be another `gwindow` instance. In this case, the new window will be positioned over the specified window and be transient for the window. That is, it will be disposed when the

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

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

3. gWidgets: Container Widgets

parent window is. This is useful, say, when a main window opens a dialog window to gather values.

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

```
child_window <- gwindow("A child window", parent = window,
                        width = 200, height = 200)
```

Handlers Windows objects can be closed programmatically through their `dispose` method. Windows may also be closed through the window manager with the click of a close icon in the title bar. The default event is the close event. For example, the following will pop up any error messages through a call to `galert` until the window is closed:

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

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

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

Clicking the button will signal an error, and the error handler will display an alert pop-up. (This last part fails under `tcltk` due to that package's handling of errors in callbacks.)

The handler argument is called just before the window is destroyed, but one 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:

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

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 box container with a text label.
<code>gexpandgroup</code>	Creates a box container with a label and a trigger to expand/collapse.
<code>glayout</code>	Creates a grid container.
<code>gpanedgroup</code>	Creates a container for two child widgets with a handle to assign allocation of space.
<code>gnotebook</code>	Creates a tabbed notebook container for holding a collection of child widgets.

In most GUIs, the use of menu bars, toolbars, and status bars is often reserved for the main window, while dialogs are not decorated so. In `gWidgets` it is suggested, although not strictly enforced unless done so by the underlying toolkit, that these be added only to a top-level window. We discuss these widgets later in Section 4.7.

A modal window

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

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

This dialog is used in a slightly different manner, requiring the use of a call to `visible` (not `visible<-`). There are three basic steps: an initial call to `gbasicdialog` to return a container to be used as the parent container for a child component, a construction of the dialog, then a call to the `visible` method on the dialog with `set=TRUE` specified. The dialog is closed through clicking one of its buttons, through a window manager event, or programmatically through its `dispose` method.

In Example 4.6 we define a GUI to assist with the task of collapsing factor levels. This wrapper function is used:

```
collapseFactor <- function(fac, parent = NULL) {
  out <- character()
  window <-
    gbasicdialog("Collapse factor levels", parent = parent,
```

Table 3.2: Container methods.

Method	Description
<code>add</code>	Adds a child object to a parent container. Called when a parent container is specified to the <code>container</code> argument of the widget constructor, in which case the <code>...</code> arguments are passed to this method.
<code>delete</code>	Removes a child object from a parent container.
<code>dispose</code>	Destroys container and children.
<code>enabled<-</code>	Sets sensitivity of child components.
<code>visible<-</code>	Hides or shows child components.

```
        handler = function(h,...) {
            new_fac <- relevel_factor$get_value()
            out <- factor(new_fac)
        })
group <- ggroup(cont = window)
relevel_factor <- CollapseFactor$new(fac, cont = group)
visible(window, set = TRUE)
out
}
```

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

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

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:

```
window <- gwindow("Some buttons", visible = FALSE)
```

```
group <- ggroup(horizontal = TRUE, cont = window)
cancel_button <- gbutton("cancel", cont = group)
ok_button <- gbutton("ok", cont = group)
visible(window) <- TRUE
```

The add method When packing in child widgets, the `add` method is used. In our example above, this is called by the `gbutton` constructor when the `container` argument is specified.⁴ Unlike with the underlying graphical toolkits, there is no means to specify other styles of packing, such as from the ends, or in the middle, by some index.

The `add` method for box containers has a few arguments to customize where the child widgets are placed and how they respond when their parent window is resized. These are passed through the `...` argument of the constructor. Figure 3.2 shows some difference in how these arguments are implemented.⁵

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

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

If a widget does not expand or if it does but does not fill in both directions, it can be anchored into its available space in more than one position. The `anchor` argument can be specified to suggest where to anchor the child. It takes a numeric vector representing Cartesian

⁴In this text, the `add` method is typically called from the constructor, but there are two cases for which one calls it directly. The first is if one wishes to integrate a widget from the underlying graphical toolkit into a `gWidgets` GUI. An example where the `tkrplot` package is embedded in a GUI is given in Section 5.1. The second case is when a widget is removed from a GUI through `delete`. In most cases it may be added back in with `add`.

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

⁶For `GTK+`, filling always occurs orthogonally to the direction of packing. This is why the top and bottom buttons (when `expand=FALSE`) in Figure 3.2 for `gWidgetsRGtk2` stretch across the container. To avoid this filling, pack the button in a horizontal `ggroup` container.



Figure 3.2: The `expand`, `fill`, and `anchor` arguments are implemented slightly differently in the different packages. (`gWidgetsRGtk2` on left, `gWidgetstcltk` in middle, and `gWidgetsQt` on right.). For `GTK+`, child components packed in a box container always fill in the direction opposite the packing, in this case the “`x`” direction. As such, the `anchor` directive has no effect. For `tcltk` a widget fills only if `expand=TRUE` is given. For `gWidgetsQt` expansion and fill are linked together.

coordinates (length two), with either value being `-1`, `0`, or `1`. For example, a value of `c(1,1)` would specify the northwest corner.

Deleting components The `delete` method can be used to remove a child component from a container. In some toolkits, this child may be added back at a later time (with `add`), but this isn’t part of the API. In the case where you wish to hide a child temporarily, its `visible<-` method can usually be used, although some widgets give this method a different meaning.⁷

Spacing For spacing between the child components, the constructor’s argument `spacing` may be used to specify, in pixels, the amount of space between the child widgets. For `ggroup` instances, this can later be set through the `svalue` method. The method `addSpace` can add a non-uniform amount of space between two widgets packed next to each other, whereas the method `addSpring` will place an invisible spring between two widgets, forcing them apart. Both are useful for laying out buttons. We used a spring before the “source” button for the GUI in Figure 3.1 to push it to the right.

For example, we might modify our button layout example to include a “help” button on the far left and the other buttons on the right with a fixed amount of space between them as follows (Figure 3.3):

```

window <- gwindow("Some buttons", visible = FALSE)
group <- ggroup(horizontal = TRUE, spacing = 6, cont = window)
help_button <- gbutton("help", cont = group)
addSpring(group)

```

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

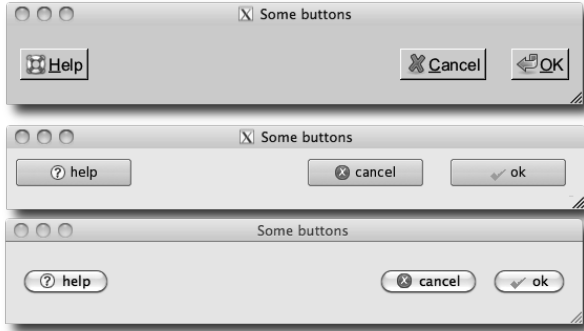


Figure 3.3: Button layout for RGtk2 (top), tcltk (middle), and qtbase (bottom). Although the same code is used for each, the different styling yields varying sizes.

```
cancel_button <- gbutton("cancel", cont = group)
addSpace(group, 12)                                # 6 + 12 + 6 pixels
ok_button <- gbutton("ok", cont = group)
visible(window) <- TRUE
```

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

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

The `gframe` and `gexpandgroup` containers

We discuss briefly two widgets that provide the same interface as `ggroup`. Much of the previous discussion applies.

Framed containers are used to link the child elements visually using a border and label. The `gframe` constructor produces them. In Figure 3.1 the table to select the file is nested in a frame to give the user some indication as to what to do.

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

The basic framed container is used along these lines:

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

Expandable containers are useful when their child items need not be visible all the time. The typical design involves a trigger icon with an accompanying label indicating to the user that a click can disclose or hide some additional information.⁸ This class overrides the `visible<-` method to initiate the hiding or showing of its child area, not the entire container.

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

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

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

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

3.3 Grid layout: the `glayout` container

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

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

```
window <- gwindow("glayout example", visible = FALSE)
lyt <- glayout(cont = window, spacing = 5)
right <- c(1,0); left <- c(-1,0)
lyt[1,1, anchor = right] <- "name"
lyt[1,2, anchor = left ] <- gedit("George Washington",
```

⁸How each toolkit resizes after a the child widget collapses varies, so using this container can cause layout issues if cross-toolkit portability is an issue.

```

      cont = lyt)
#
lyt[2,1, anchor = right] <- "rank"
lyt[2,2, anchor = left ] <- gedit("General", cont = lyt)
#
lyt[3,1, anchor = right] <- "serial number"
lyt[3,2, anchor = left ] <- gedit("1", cont = lyt)
visible(window) <- TRUE

```

When adding a child, in addition to being on the left-hand side of the [`<-`] call, the `glayout` container should be specified as the widget's parent container.⁹ For convenience, if the right-hand side is a string, a label will be generated. To align a widget within a cell, the `anchor` argument of the [`<-glayout`] method is used. The example above illustrates how this can be used to achieve a center balance.

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

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

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

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

```
[1] "George Washington" "General" "1"
```

3.4 Paned containers: the `gpanedgroup` container

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

⁹This 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.

To add children, the container should be passed as the parent during the construction of each of the two child widgets. These might be other container constructors, which is the typical usage for more complicated layouts.

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

A simplified version of the layout code in Figure 3.1 would be

```
examples <- system.file("Examples", "ch-gWidgets",
                        package = "ProgGUIinR")
files <- list.files(examples)
#
window <- gwindow("gpanedgroup example", visible = FALSE)
paned <- gpanedgroup(cont = window)
tbl <- gtable(files, cont = paned) # left side
txt_widget <- gtext("", cont = paned, expand = TRUE) # right
visible(window) <- TRUE
svalue(paned) <- 0.33 # after drawing
```

3.5 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 constructor has a few arguments, not all supported by each toolkit. The argument `tab.pos` is used to specify the location of the tabs by assigning a value of 1 through 4, with 1 being the bottom, 2 the left side, 3 the top, and 4 the right side, with the default being 3 (similar numbering as used in par). The `closebuttons` argument takes a logical value indicating whether the tabs should have close buttons on them. In this case, the argument `dontCloseThese` can be used to specify which tabs, by index, should not be closable.

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

The `svalue` method returns the index of the currently raised tab, whereas `svalue<-` can be used to switch the page to the specified tab. The currently shown tab can be removed using the `dispose` method. To remove a different tab, use this method in combination with `svalue<-`.

(When removing many tabs, you will want to start from the end, as otherwise the tab positions change during removal.)

From some viewpoint, the notebook widget is viewed as a vector of child widgets, named according to the tab labels. As such, the `[]` method returns the child components (by index), the `names` method refers to the tab names, and the `length` method returns the number of pages held by the notebook.

Example 3.1: 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:

```
window <- gwindow("gnotebook example")
notebook <- gnotebook(cont = window)
```

New pages are added as follows:

```
add_a_page <- function(file_name) {
  f <- system.file(file_name, package = "ProgGUIinR")
  gtext(paste(readLines(f), collapse="\n"),
        cont = notebook, label = file_name)
}
add_a_page("DESCRIPTION")
```

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

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

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

```
svalue(notebook) <- length(notebook)
```

To remove the current page:

```
dispose(notebook)
```

This page intentionally left blank

gWidgets: Control Widgets

This chapter discusses the basic GUI controls provided by `gWidgets`. We defer discussion of the R-specific widgets to the next chapter.

4.1 Buttons

The button widget allows a user to initiate an action through clicking on it. Buttons have labels, conventionally verbs, indicating action, and often icons. The `gbutton` constructor has an argument `text` to specify the text. For text that matches the stock icons of `gWidgets` (Section 4.2), an icon will appear. (The `ok` button below, but not the `parButton` one.)

In common with the other controls, the argument `handler` is used to specify a callback, and the `action` argument will be passed along to this callback (unless it is a `gaction` object, whose case is described in Section 4.7). The default handler is the `clickHandler`, 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.

```

window <- gwindow("Make a plot")
group <- ggroup(horizontal = FALSE, cont = window)
glabel("... Fill me in ...", cont = group)
button_group <- ggroup(cont = group)
addSpring(button_group)
parButton <- gbutton("par (mfrow) ...", cont = button_group)

```

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

```

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

```

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

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

```

lyt[3,2] <-
  gbutton("ok", cont = lyt, handler =
    function(h,...) {
      x <- as.numeric(c(svalue(nr), svalue(nc)))
      par(mfrow = x)
      dispose(child)
    })
  })

```

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

4.2 Labels

The `glabel` constructor produces a basic label widget. We've already seen its use in a number of examples. The main property, the label's text, is specified through the `text` argument. This is a character vector of length 1 or is coerced into one by collapsing the vector with newlines. The `svalue` method will return the label text as a single string, whereas the `svalue<-` method is available to set the text programmatically. The `font<-` method can also be used to set the text markup (Table 4.3).¹

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

```

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

```

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

HTML text

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

Status bars

In `gWidgets`, status bars are simply labels placed at the bottom of a top-level window to leave informative, but non-disruptive, messages for the user. The `gstatusbar` constructor provides this widget. The `container` argument should be a top-level window instance. The only property is the

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

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

Icons and images

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

Other graphic files and the stock icons can be displayed by the `gimage` widget.² The file to display is specified through the `filename` argument of the constructor. This value is combined with that of the `dirname` argument to specify the file path. Stock icons are specified by using their name for the `filename` argument and the character string "stock" for the `dirname` argument.³

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

The default handler is a button-click handler.

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

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

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

Example 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 (Figure 4.1) to select a color from, as an alternative to a more complicated-color chooser dialog.⁴

We begin by defining 16 arbitrary colors.

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

³For `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".

⁴If `gWidgetstcltk` is used the image files would need to be converted to gif format, as png format is not a natively supported image type.

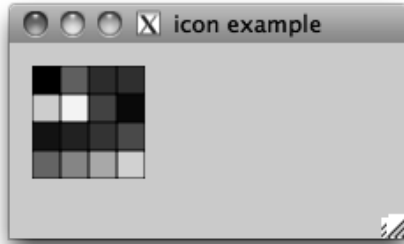


Figure 4.1: A table of stock icons created on the fly.

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

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

```
require(grid)
icon_dir <- tempdir(); iconSize <- 16;
make_color_icon <- function(i) {
  filename <- file.path(icon_dir,
                        sprintf("color-%s.png", i))
  png(file = filename, width = iconSize, height = iconSize)
  grid.newpage()
  grid.draw(rectGrob(gp = gpar(fill = i)))
  dev.off()
  return(filename)
}
```

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

```
icons <- sapply(some_colors, make_color_icon)
icon_names <- sprintf("color-%s", some_colors)
addStockIcons(icon_names, 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.

```
window <- gwindow("Icon example", visible=FALSE)
callback <- function(h,...) galert(h$action, parent = window)
lyt <- glayout(cont = window, spacing = 0)
for(i in 1:4) {
  for(j in 1:4) {
    ind <- (i - 1) * 4 + j
```



```
    lyt[i,j] <- gimage(icons[ind], handler = callback,
                      action = icon_names[ind], cont = lyt)
  }
}
visible(window) <- TRUE
```

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

4.3 Text-editing controls

The `gWidgets` package, following the underlying toolkits, has two main widgets for editing text: `gedit` for a single line of editable text and `gtext` for multiline, editable text. Each is simple to use but provides much less flexibility than is possible with the toolkit widgets.

Single-line, editable text

The `gedit` constructor produces a widget to display a single line of editable text. The main property is the `text`, which can be set initially through the `text` argument. If not specified, and the argument `initial.msg` is, then this initial message is shown until the widget receives the focus to guide the user. If it is desirable to set the width of the widget, the `width` argument allows the specification in terms of number of characters allowed to display without horizontal scrolling. The width of the widget may also be specified in pixel size through the `size<-` method.

A simple usage might be:

```
window <- gwindow("Simple gedit example", visible = FALSE)
group <- ggroup(cont = window)
entry <- gedit("", initial.msg = "Enter your name...",
              cont = group)
visible(window) <- TRUE
```

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

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

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

For example, the following can be used to collect one of the fifty state names in the United States:

```
window <- gwindow("gedit example", visible = FALSE)
group <- ggroup(cont = window)
glabel("State name:", cont = group)
entry <- gedit("", cont = group)
entry[] <- state.name
visible(window) <- TRUE
```

Handlers The default handler for the `gedit` widget is called when the text area is activated by the return key being pressed. Use `addHandlerBlur` to add a callback for the event of losing focus. The `addHandlerKeystroke` method can assign a handler to be called when a key is released. For the toolkits that support it, the specific key is given in the `key` component of the list `h` (the first argument).⁵

Example 4.2: Validation

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

```
window <- gwindow("Validation example")
lyt <- glayout(cont = window)
lyt[1,1] <- "R expression:"
lyt[1,2] <- (entry <- gedit("", cont = lyt))
```

We use the `evaluate` package to see if the expression is valid.⁶

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

⁶The basic way to evaluate an R expression given as a string is to use the combination of `eval` and `parse`, as in `eval(parse(text=string))`. The resulting output can usually be

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

We validate our expression when the user commits the change by pressing the return key while the widget has focus.

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

Multiline, editable text

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

The `svalue` method retrieves the text stored in the buffer. If the argument `drop=TRUE` is specified, then only the currently selected text will be returned. Text in multiple lines is returned as a single string with `"\n"` separating the lines.

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

Fonts Fonts can be specified for the entire buffer or the selection using the specifications in Table 4.3. To specify fonts for the entire buffer, use the `font.attr` argument of the constructor. The `font<-` method serves the same purpose, provided there is no selection when it is called. If there is a

captured with the `capture.output` function. However, there can be errors: parse errors or otherwise. A few packages provide functions to assist with this task, notably the `evaluate` function in the same-named `evaluate` package, and the `parseText` and `captureAll` functions in the `svMisc` package. We illustrate both in this part of the text.

Table 4.2: Possible specifications for setting font properties. Font values of an object are changed with named lists, as in `font(obj)<-list(weight="bold", size=12, color="red")`.

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

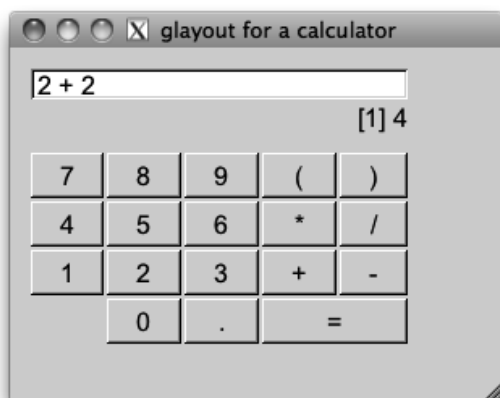


Figure 4.2: Dumbing down R with `gWidgets` to make a calculator interface.

selection, the font change will be applied only to the selection. Finally, the `font.attr` argument for the `insert` method specifies the font attributes for the inserted text.

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

Example 4.3: A calculator

This example shows how we might use the widgets just discussed to make a GUI that resembles a calculator (Figure 4.2). Such a GUI may offer familiarity to new R users, although certainly it is no replacement for a command line.

The `layout` container is used to arrange the widgets neatly. 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: 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, "+" , "-" ))
#
window <- gwindow("glayout for a calculator", visible = FALSE)
group <- ggroup(cont = window, expand = TRUE, horiz = FALSE)
lyt <- glayout(cont = group, spacing = 2)
lyt[1, 1:5, anchor = c(-1,0)] <- # span 5 columns
  (eqn_area <- gedit("", cont = lyt))
lyt[2, 1:5, anchor = c(1,0)] <-
  (output_area <- glabel("", cont = lyt))
#
button_list <- list()
for(i in 3:5) {
  for(j in 1:5) {
    val <- buttons[i-2, j]
    lyt[i,j] <- (button_list[[val]] <- gbutton(val, cont=lyt))
  }
}
lyt[6,2] <- (button_list[["0"]] <- gbutton("0", cont = lyt))
lyt[6,3] <- (button_list[["."]] <- gbutton(".", cont = lyt))
lyt[6,4:5] <- (eq_button <- gbutton(" = ", cont = lyt))
#
visible(window) <- TRUE
```

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

```
add_button <- function(h, ...) {
  cur_expr <- svalue(eqn_area)
  new_char <- svalue(h$obj) # the button's value
  svalue(eqn_area) <- paste(cur_expr, new_char, sep = "")
  svalue(output_area) <- "" # clear label
}
sapply(button_list, addHandlerChanged, handler = add_button)
```

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

```
require(evaluate)
addHandlerClicked(eq_button, handler = function(h,...) {
  curExpr <- svalue(eqn_area)
  out <- try(evaluate:::evaluate(curExpr), silent = TRUE)
```

```

if(inherits(out, "try-error")) {
  galert("Parse error", parent = eq_button)
} else if(is(out[[2]], "error")) {
  msg <- sprintf("Error: %s", out[[2]]$message)
  galert(msg, parent = eq_button)
} else {
  svalue(output_area) <- out[[2]]
  svalue(eqn_area) <- "" # restart
}
})

```

4.4 Selection controls

A common task for a GUI control is to select a value or values from a set of numbers or a table of numbers. Figure 4.3 shows a simple GUI for the EBImage package allowing a user to adjust a few of the image properties using various selection widgets. Although it is unlikely we would use R for such a task, as opposed to Gimp, say, we use this example, as the mapping between controls and actions should be familiar.

In gWidgets the abstract view for selection widgets is that the user is choosing from a set of items stored as a vector (or data frame). The familiar R methods are used to manipulate this underlying data store. The controls in gWidgets that display such data have the methods `[], [<-, length, dim, names, and names<-`, as appropriate. The `svalue` method then refers to the user-selected value. This selection may be a value or an index, and the `svalue` method has the argument `index` to specify which.

This section discusses several such selection controls, which 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 whether the widget should initially be checked. The default is `TRUE` (there is no third, uncommitted state as is possible with some toolkits). By default the label will be drawn aside a box that the user can check. If the argument `use.togglebutton` is `TRUE`, a toggle button – which appears depressed when `TRUE` – is used instead.

In Figure 4.3, a toggle button is used for “Thresh” and could be constructed as

```

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

```

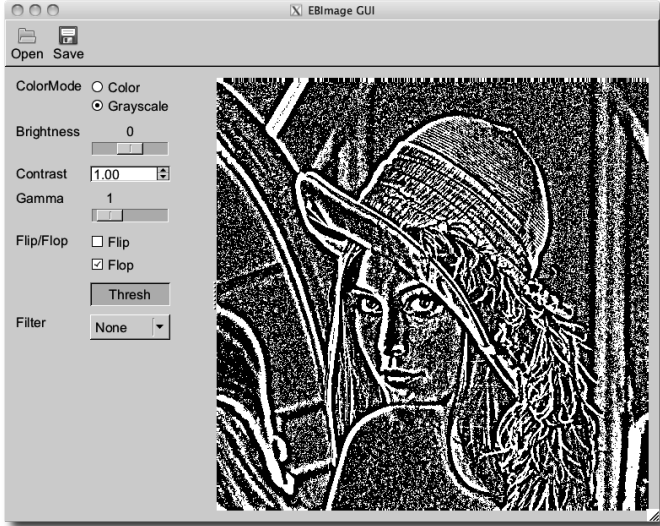


Figure 4.3: A simple GUI for the EBImage package illustrating many selection widgets.

The `svalue` method returns a logical indicating whether 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 when the state toggles. If it is desired that the handler be called only in the TRUE state, say, we need to check within the handler for this. For example:

```

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

Radio buttons

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 (two or more). These items may be displayed horizontally or vertically (the default) as specified by the `horizontal` argument, which expects a logical. The selected argu-

ment specifies the initially selected item, by index, with a default of the first.

In Figure 4.3 a radio button is used for `ColorMode` and could be constructed as

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

The currently selected item is returned by `svalue` as the label text or by the index if the argument `index` is `TRUE`. The item may be set with the `svalue<-` method. Again, the item may be specified by the label or by an index, the latter when the argument `index=TRUE` is specified.

The data store is the set of labels and may be respecified with the `[<-` method.

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

A group of checkboxes

A group of checkboxes is produced by the `gcheckboxgroup` constructor. This convenience widget is similar to a radio group, except that it allows the selection of none, one, or more than one of a set of items. The `items` argument is used to specify the values. The state of whether an item is selected can be set with a logical vector of the same size as the number of items to the `checked` argument; recycling is used. The item layout can be controlled by the `horizontal` argument. The default is a vertical layout (`horizontal=FALSE`).

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

In Figure 4.3 a group of check boxes is used to allow the user to flip or flop the image. It could be created with

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

The current selection is retrieved as a character vector through the `svalue` method. The `index=TRUE` argument instructs `svalue` to return the selected indices instead. These are 0-length if no selection is made. As a checkbox group is like both a checkbox and a radio button group, we can set the selected values three different ways. As with a checkbox, the selected values can be set by specifying a logical vector through the `svalue<-`

method. As with radio-button groups, the selected values can also be set with a character vector indicating which labels should be selected, or if `index=TRUE` is given, using a numeric index vector.

That is, each of these has the same effect:

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

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

A combo box

Combo boxes are constructed by `gcombobox`.⁷ As with the other selection widgets, the choices are specified to the argument `items`. However, this may be a vector of values or a data frame in which the first column defines the choices. For toolkits that support icons in the combo box widget, if the data is specified as a data frame, the second column may be used signify a stock icon to decorate the selection, and, by design (but implemented only for `gWidgetsQt`), a third column specifies a tooltip to appear when the mouse hovers over a possible selection.

The combo box in Figure 4.3 could be coded with:

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

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

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

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

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

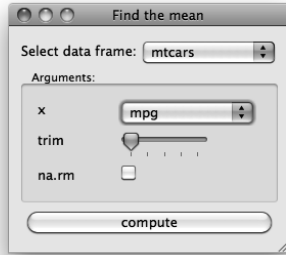


Figure 4.4: GUI used to collect arguments for a call to `mean.default`.

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

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

Example 4.4: Updating combo boxes

A common feature in many GUIs is to have one combo box update another once a selection is made. The following example employs this design to create a simple GUI for collecting the arguments for computing the mean of a numeric variable (Figure 4.4).

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

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

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

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

```
window <- gwindow("Find the mean", visible = FALSE)
```

```

group <- ggroup(cont = window, horizontal = FALSE)
group1 <- ggroup(cont = group)
glabel("Select data frame:", cont = group1)
df_combo_box <- gcombobox(avail_DFs(), cont = group1)
##
frame <- gframe("Arguments:", cont = group, horizontal=FALSE)
enabled(frame) <- FALSE
lyt <- glayout(cont = frame, expand = TRUE)
widget_list <- list()
##
lyt[1,1] <- "x"
lyt[1,2] <- (widget_list$x <- gcombobox("          ",
                                       cont = lyt))
##
lyt[2,1] <- "trim"
lyt[2,2] <-
  (widget_list$trim <- gslider(from = 0, to = 0.5, by = 0.01,
                              cont = lyt))
##
lyt[3,1] <- "na.rm"
lyt[3,2] <-
  (widget_list$na.rm <- gcheckbox("", checked = TRUE,
                                cont = lyt))
group2 <- ggroup(cont = group)
compute_button <- gbutton("compute", cont = group2)

```

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

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

```

addHandlerChanged(df_combo_box, handler = function(h,...) {
  val <- svalue(h$obj)
  enabled(frame) <- val != ""
  enabled(compute_button) <- val != ""
  if(val != "")
    widget_list$x[] <- get_numeric(val)
  svalue(widget_list$x, index = TRUE) <- 0
})

```

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

```

addHandlerChanged(compute_button, handler = function(h,...) {

```

```

out <- lapply(widget_list, svalue)
out$x <- get(out$x, get(svalue(df_combo_box),
                       envir = .GlobalEnv))
print(do.call(mean.default, out))
})

```

A slider control

The `gslider` constructor creates a scale widget that allows the user to select a value from the specified sequence. The basic arguments mirror that of the `seq` function in R: `from`, `to`, and `by`. However, if `from` is a vector, then it is assumed it presents an ordered sequence of values from which to select. In addition to the arguments to specify the sequence, the argument `value` is used to set the initial value of the widget, and `horizontal` controls how the slider is drawn, `TRUE` for horizontal, `FALSE` for vertical.

In Figure 4.3, a slider is used to update the brightness. The call is similar to:

```

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

```

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

In Figure 4.3, the `gWidgetsRGtk2` package is used. That toolkit shows a tooltip with the current value; for other toolkits the slider implementation does not show the value. We can add a label to show this (or combine the slider with a spin button). Adding a label follows this pattern:

```

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

```

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

A spin button control

The spin button control constructed by `gspinbutton` is similar to `gslider` when used with numeric data but presents the user a more precise way to select the value. The `from`, `to`, and `by` arguments must be specified. The argument `digits` specifies the number of digits displayed.

In Figure 4.3 a spin button is used to adjust the contrast, which expects a numeric value. The following will reproduce it:

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

Selecting from the file system

The `gfile` dialog allows us 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 with a button that initiates this selection.

The “Open” button in Figure 4.3 is bound to this action:

```
f <- gfile("Open an image file",
          type="open",
          filter=list("Image file" = list(
                        patterns = c("*.gif", "*.jpg", "*.png")
                      ),
                    "All files" = list(patterns = c("*"))
          ))
if(!is.na(f))
  readImage(f) ## ...
```

The selection type is specified by the `type` argument with values of `open` to select an existing file, `save` to select a file to write to, and `selectdir` to select a directory. The `filter` argument is toolkit dependent. For `RGtk2`, the `filter` argument used above will filter the possible selections. The dialog returns the path of the file, or `NA` if the dialog was canceled.

Although working with the return value is easy enough, if desired, we can specify a handler to the constructor to call on the file or directory name. The component `file` of the first argument to the handler contains the file name.

Selecting a date

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

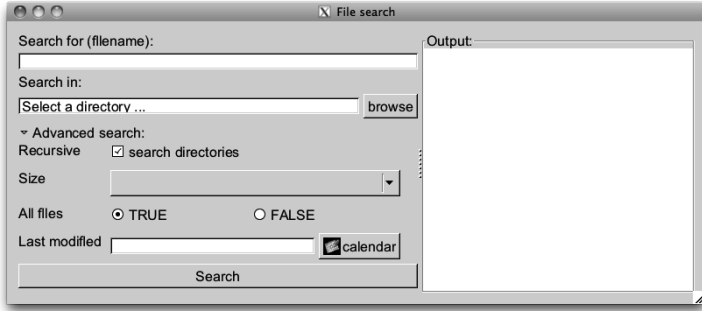


Figure 4.5: File-search dialog showing advanced search features.

widget. The argument `text` argument specifies the initial text. The format of the date is specified by the `format` argument.

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

Example 4.5: Selecting from a file system

We return to the file-selection GUI used as an example in Chapter 2. Our goal here is to add more features to support advanced searching. Imagine we have a function `file_search`, which in addition to arguments for a pattern and directory, has arguments modified to pass a date string; size to pass a descriptive `small`, `medium` or `large`; and an argument `visible` to indicate whether all files (including dot files) should be looked at.

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

```
adv_search <- gexpandgroup("Advanced search:",
                          cont = nested_group)
visible(adv_search) <- FALSE
lyt <- glayout(cont = adv_search)
lyt[1,1] <- "Recursive"
lyt[1,2] <- (adv_rec <-
  gcheckbox("search directories", checked = TRUE, cont=lyt))
lyt[2,1] <- "Size"
lyt[2,2] <- (adv_size <-
  gcombobox(c("", "small", "medium", "large"), cont = lyt))
lyt[3,1] <- "All files"
lyt[3,2] <- (adv_visible <-
  gradio(c(TRUE, FALSE), horizontal = TRUE, cont = lyt))
```

```
lyt[4,1] <- "Last modified"
lyt[4,2] <- (adv_modified <-
             gcalendar("", format = "%Y-%m-%d", cont = lyt))
```

As can be seen (Figure 4.5), we use a grid layout and a mix of the controls offered by gWidgets.

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

```
addHandlerChanged(search_btn, handler=function(h,...) {
  pattern <- glob2rx(svalue(txt_pattern))
  start_at <- svalue(start_dir)
  modified <- NULL
  size <- NULL

  ## from advanced
  subfolders <- svalue(adv_rec)
  visible <- svalue(adv_visible)
  if((tmp <- svalue(adv_size)) != "") size <- tmp
  if(!is.na(tmp <- svalue(adv_modified))) modified <- tmp

  ## function call
  file_names <- file_search(pattern, start_at, subfolders,
                           modified = modified,
                           size = size, visible = visible)
  dispose(search_results) # clear
  if(length(file_names))
    svalue(search_results) <- file_names
  else
    galert("No matching files found", parent = window)
})
```

4.5 Display of tabular data

The `gtable` constructor⁸ produces a widget that displays data in a tabular form from which the user can select one (or more) rows. The performance under `gWidgetsRGtk2` and `gWidgetsQt` is much faster and able to handle larger data stores than under `gWidgetstcltk`, as there is no enhanced data frame model in Tcl/Tk. At a minimum, all perform well on moderate-sized data sets (ten or so columns and fewer than 500 rows).⁹

⁸The `gtable` widget shows clearly the tradeoffs between using `gWidgets` and a native toolkit under R. As will be seen in later chapters, setting up a table to display a data frame using the toolkit packages directly can involve a fair amount of coding as compared to `gtable`, which makes it very easy. However, `gWidgets` provides far less functionality. For example, there is no means to adjust the formatting of the displayed text, or to embed other widgets, such as check boxes, into the tabular display.

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

The data is specified through the `items` argument. This value may be a data frame, matrix, or vector. Vectors and matrices are coerced to data frames, with `stringsAsFactors=FALSE`. The data is presented in a tabular form, with column headers derived from the `names` attribute of the data frame (but no row names). The `items` argument can be a 0-row data frame, but the column classes must match the eventual data to be used.

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

```
window <- gwindow("gtable example")
DFs <- gtable(ProgGUIinR:::avail_dfs(), cont = window)
```

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

Icons The `icon.FUN` argument can be used to place a stock icon in a left-most column. This argument takes a function of a single argument – the data frame being shown – and should return a character vector of stock icon names, one for each row.

Selection Users can select by case (row) – not by observation (column) – from this widget. The actual value returned by a selection is controlled by the constructor’s argument `chosenCol`, which specifies which column’s value will be returned for the given index, as the user can specify only the row. The `multiple` argument can be specified to allow the user to select more than one row.

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

The underlying data store is referenced by the `[]` method. (That is matrix-like access is supported, but not list-like.) Indices can be used to access a slice. Values can be set using the `[<-` method, but be warned that this method is not as flexible as assigning to a data frame. The underlying toolkits may not like to change the type of data displayed in a column or reduce the number of columns displayed, so when updating a column do not assume some underlying coercion, as can happen with R’s data frames. (This is why the initial `items`, even if specified through

a 0-row data frame, need to be of the correct class.) To replace the data store, the `[<-` can be used, as with `obj[] <- new_data_frame`. The methods `names` and `names<-` refer to the column headers, and `dim` and `length` the underlying dimensions of the data store.

To update the list of data frames in our DFs widget, we can define a function such as the following:

```
updateDfs <- function() {
  DFs[] <- ProgGUIinR:::avail_dfs()
}
```

Handlers Selection is done through a single click. The `addHandlerClick` method can be used to assign a handler to a change of selection event. 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)
  print(summary(get(val, envir = .GlobalEnv))) # some action
})
```

Example 4.6: Collapsing factors

A somewhat tedious task in R is the recoding or collapsing of factor levels. This example provides a GUI to facilitate this. In Section 3.1 we provided a function to wrap this GUI within a modal dialog.

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

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

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

This `make_gui` function does the hard work. (Figure 4.6 shows a screenshot.) We have just two widgets, placed in a paned group. The one on the left is a table that displays two columns: the old values and the collapsed or recoded values. The widget on the right is a combo box for entering a

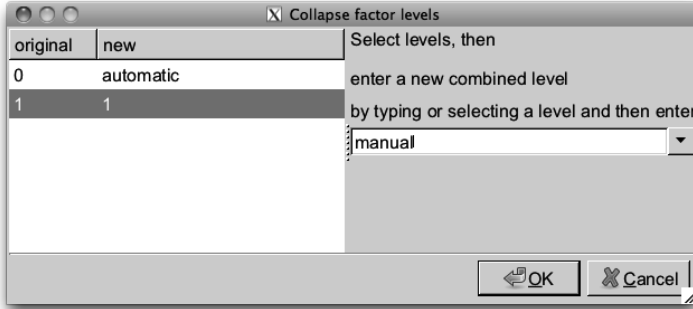


Figure 4.6: A GUI to facilitate the recoding or collapsing of a factor's levels. For this, we select the desired levels to rename or collapse, then enter a new label on the right. Activating the combo box will update the "new" column on the left.

new factor level or selecting an existing level. The handler on the combo box updates the second column of the table to reflect the new values. We block any handler calls to avoid a loop when we set the index back to 0.

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

```
if(length(ind) == 0)
  return()
#
tbl[ind,2] <- svalue(combo_box)
svalue(tbl, index = TRUE) <- 0
blockHandler(combo_box)
combo_box[] <- sort(unique(tbl[,2]))
svalue(combo_box, index = TRUE) <- 0
unblockHandler(combo_box)
})
}
```

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

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

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

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

Filtering The arguments `filter.column` and `filter.FUN` allow us to specify whether the user can filter, or limit, the display of the values in the data store. The simplest case is if a column number is specified to the `filter.column` argument. In this case a combo box is added to the widget with values taken from the unique values in the specified column. Changing the value of the combo box restricts the display of the data to just those rows in which the value in the filter column matches the combo box value. More advanced filtering can be specified using the `filter.FUN` argument. If this is a function, then it takes arguments (`data_frame`, `filter.by`) where the data frame is the data, and the `filter.by` value is the

Package	Version	Priority	Depends
MLecens	0.1-3	NA	NA
USCensus2000	0.09	NA	R (>= 2.10), mapprotools, sp, foreign, rstats, utils, USCensus2000tract, USCensus2000cdp, gpcplib
USCensus2000add	0.07	NA	R (>= 2.10), mapprotools, sp, foreign, rstats, utils, USCensus2000blkgrp, USCensus2000cdp, gpcplib, XML, US

Figure 4.7: Example of using a filter to narrow the display of tabular data.

state of a combo box whose values are specified through the argument `filter.labels`. This function should return a logical vector with length matching the number of rows in the data frame. Only rows corresponding to TRUE values will be displayed.

If `filter.FUN` is the character string “manual” then the `visible<-` method can be used to control the filtering, again by specifying a logical vector of the proper length. See Example 4.8 for an application.

Example 4.7: Simple filtering

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

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

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 4.8: More complex filtering

Even with moderate-sized data sets, the number of rows can be quite large, in which case it is inconvenient to use a table for selection unless some means of searching or filtering the data is used. This example displays the many possible CRAN packages to show how a `gedit` instance can be

used as a search box to filter the display of data (Figure 4.7). The `addHandlerKeystroke` method is used, so that the search results are updated as the user types.

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

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

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

```
window <- gwindow("test of filter")
group <- ggroup(cont = window, horizontal = FALSE)
entry <- gedit("", cont = group)
tbl <- gtable(avail_pkgs, cont = group, filter.FUN = "manual",
             expand = TRUE)
```

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

Different search criteria may be desired, so it makes sense to separate 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.)

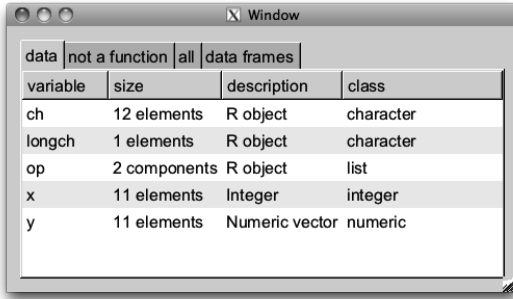
```
our_match <- function(cur_val, vals) {
  grepl(cur_val, vals)
}
```

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

```
id <- addHandlerKeystroke(entry, handler = function(h, ...) {
  vals <- tbl[, 1, drop = TRUE]
  cur_val <- svalue(h$obj)
  visible(tbl) <- our_match(cur_val, vals)
})
```

Example 4.9: Using the "observer pattern" to write a workspace view

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



variable	size	description	class
ch	12 elements	R object	character
longch	1 elements	R object	character
op	2 components	R object	list
x	11 elements	Integer	integer
y	11 elements	Numeric vector	numeric

Figure 4.8: A notebook showing various views of the objects in the global workspace. The example uses the Observer pattern to keep the views synchronized.

This pattern in which a central, dynamic source of data is to used and shared amongst many different pieces of a GUI, is a common one. To address the complexity that arises as the components of a GUI get more intertwined, standard design patterns have been employed. For this task, the *observer pattern* is often used. This pattern is defined in *Head First Design Patterns*^[7] to describe a one-to-many relationship among a set of objects where when the state of one object changes, all of its dependents are notified.

Figure 4.9 shows a class diagram of the two different types of objects involved:

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

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

The package `objectSignals` provides a comprehensive implementation of this pattern. Based on that, the `objectProperties` package implements *properties*: fields with enhanced functionality, including observability. These

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

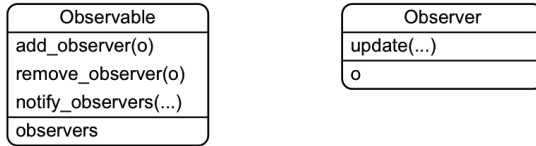


Figure 4.9: Observable and observer classes and their basic methods. An observable object may have many observers, which are notified through their update method when a change is made.

are similar to the properties that we will see later in the GTK+ and Qt libraries. We use the `properties` function to create an “observable” property and `connect`, from `objectSignals`, to add an observer. When the property is changed, any observers are notified.

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

```
library(objectProperties)
require(digest)
WSModel <- setRefClass("WSModel",
  fields = c(
    properties(list(ws_objects = "character")),
    ws_objects_digests = "character"
  ))
```

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

```
WSModel$methods(
  .get_objects_digests = function() {
    "Helper function to return list with names, digests"
    items <- ls(envir = .GlobalEnv)
    objects <- mget(items, .GlobalEnv)
    trim <- !sapply(objects, is, class2 = "refClass")
    list(items[trim],
         sapply(objects[trim], digest))
  },
```

```

initialize = function() {
  objs <- .get_objects_digests() # call helper
  initFields(ws_objects = objs[[1]],
             ws_objects_digests = objs[[2]])
  callSuper()
},
refresh = function() {
  objs <- .get_objects_digests()
  cur_objects <- objs[[1]]
  cur_digests <- objs[[2]]
  ## changes?
  if(length(cur_digests) != ws_objects_digests ||
     length(ws_objects_digests) == 0 ||
     any(cur_digests != ws_objects_digests)) {
    ws_objects <<- cur_objects # signal
    ws_objects_digests <<- cur_digests
  })
}

```

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

```

WSModel$methods(
  get = function(klass) {
    "klass a string, such as 'numeric' or '!function'"
    if(missing(klass) || length(klass) == 0)
      return(ws_objects)
    ## if we have klass, more work
    ind <- sapply(mget(ws_objects, .GlobalEnv),
                 function(x) {
                   any(sapply(klass, function(j) {
                     if(grepl("~!", j))
                       !is(x, substr(j, 2, nchar(j)))
                     else
                       is(x, j)
                   })))
                 })
    ##
    if(length(ind))
      ws_objects[ind]
    else
      character(0)
  })
}

```

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

```

WSModel$methods(
  add_observer = function(FUN, ...) {

```



```
        .self$ws_objectsChanged$connect(FUN, ...)
    })
```

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

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

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

```
WidgetView <-
    setRefClass("WidgetView",
        contains = "WSView",
        fields = list(
            klass = "character", # which classes to show
            widget = "ANY"
        ),
        methods = list(
            initialize = function(parent, model,
                klass=character(0), ...) {
                if(!missing(model)) set_model(model)
                if(!missing(parent)) init_widget(parent, ...)
                initFields(klass=klass)
                update(model)
                callSuper()
            },
            init_widget = function(parent, ...) {
                "Initialize widget"
            })
    ))
```

We write a `WidgetView` subclass to view the workspace objects using a `gtable` widget:

```
TableView <-
    setRefClass("TableView",
        contains = "WidgetView",
        methods = list(
            init_widget = function(parent, ...) {
```

```

        widget <- gtable(makeDataFrame(character(0)),
                        cont = parent, ...)
    },
    update = function(model, ...) {
        widget[] <- makeDataFrame(model$get(klass))
    })

```

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

To generate data on each object, we define some S3 classes. These are more convenient than reference classes for this task. First, we want a nice description of the size of the object:

```

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

```

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

```

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

```

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

```

makeDataFrame <- function(x, envir = .GlobalEnv) {
  DF <- data.frame(variable = character(0),
                  size = character(0),
                  description = character(0),
                  class = character(0),
                  stringsAsFactors = FALSE)

  if(length(x)) {
    l <- mget(x, envir)
    short_class <- function(x) class(x)[1]
    DF <- data.frame(variable = x,
                    size = sapply(l, size_of),
                    description=sapply(l, short_description),
                    class = sapply(l, short_class),
                    stringsAsFactors = FALSE)
  }
  DF
}

```

4. gWidgets: Control Widgets

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

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

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

```
window <- gwindow()
notebook <- gnotebook(cont = window)
##
model <- WSMModel$new()
## basic view of certain classes
view <- TableView$new(parent = notebook, model = model,
  label = "data",
  klass=c("factor","numeric", "character",
    "data.frame", "matrix", "list"))
## view of non functions
view1 <- TableView$new(parent = notebook, model = model,
  label = "not a function",
  klass = "!function"
)
## view of all
view2 <- TableView$new(parent = notebook, model = model,
  label = "all")
## a bit contrived here, but useful elsewhere
view3 <- DfView$new(parent = notebook, model = model,
  label = "data frames")
#
model$refresh()
svalue(notebook) <- 1
```

4.6 Display of hierarchical data

The `gtree` constructor can be used to display hierarchical structures, such as a file system or the components of a list. To use `gtree` we describe the tree to be shown dynamically through a function that computes the child components in terms of the path of the parent node. Although a bit more complex, this approach allows trees with many ancestors to be shown, without needing to compute the entire tree at the time of construction.

The `offspring` argument is assigned a function of two arguments, the path of a particular node and the arbitrary object passed through the optional `offspring.data` argument. This function should return a data frame with each row referring to an offspring for the node and whose first column is a key that identifies each of the offspring.

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

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

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

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

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

```
lst <- list(a = "1", b = list(a = "2", b = "3",
                             c = list(a = "4")))
window <- gwindow("Tree test")
tree <- gtree(offspring, offspring.data = lst, cont = window)
```

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

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

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

Example 4.10: Using `gtree` to explore a recursive partition

The `party` package implements a recursive partitioning algorithm for tree-based regression and classification models. The package provides an excellent plot method for the object, but in this example we demonstrate how the `gtree` widget can be used to display the hierarchical nature of the fitted object. As working directly with the return object is not for the faint of heart, such a GUI can be useful.

First, we fit a model from an example that appears in the package's vignette.

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

The `party` object tracks the hierarchical nature through its nodes. This object has a complex structure using lists to store data about the nodes. Next, we define an `offspring` function 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.

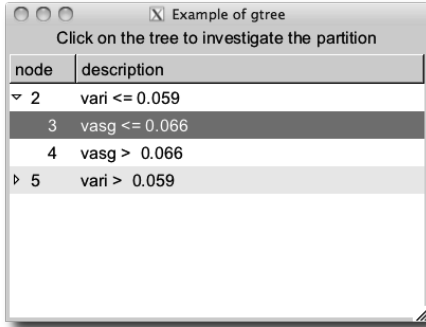


Figure 4.10: GUI to explore return value of a model fit by the party package.

For this example, the trees are all binary trees with 0 or 2 offspring, so this data frame has only 0 or 2 rows.

```

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

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

  DF <- data.frame(node = integer(2), hasOffspring=logical(2),
                  description = character(2),
                  stringsAsFactors = FALSE)

  ## party internals
  children <- c("left", "right")
  ineq <- c("<=", "> ")
  varName <- nodes(gt, node)[[1]]$psplit$variableName
  splitPoint <- nodes(gt, node)[[1]]$psplit$splitpoint

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

```

We make a simple GUI to show the widget (Figure 4.10).

```
window <- gwindow("Example of gtree")
group <- ggroup(cont = window, horizontal = FALSE)
label <- glabel("Click on the tree to investigate the
  partition", cont = group)
tree <- gtree(offspring, cont = group, 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(tree, 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, ])
  }})
```

4.7 Actions, menus, and toolbars

Actions are non-graphical objects representing an application command that is executable through one or more widgets. Actions in `gWidgets` are created through the `gaction` constructor. The arguments are `label`, `tooltip`, `icon`, `key.accel`,¹⁰ `parent`, and the standard handler and action.

The label appears as the text on a button, a menu item, or a toolbar button, whereas the icon will decorate the same, if possible. For some toolkits, the tooltip pops up when the mouse hovers. The `parent` argument is used to specify a widget whose top-level container will process the shortcut.

Methods The main methods for actions are `svalue<-` to set the label text and `enabled<-` to adjust whether the widget is sensitive to user input. All proxies of the action are set through one call. There is no method to invoke the action.

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.

```
window <- gwindow("gaction example")
action <- gaction("click me", tooltip = "Click for a message",
  icon = "ok",
  handler = function(h, ...) {
```

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

```

        print("Hello")
    },
    parent = window)
button <- gbutton(action = action, cont = window)
## .. to change
enabled(action) <- FALSE # can't click now

```

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

Toolbars

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

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

For example, first we define some actions:

```

stub <- function(h,...) gmessage("called handler",
                                parent = window)

action_list = list(
  new = gaction(label = "new", icon = "new",
                handler = stub, parent = window),
  open = gaction(label = "open", icon = "open",
                 handler = stub, parent = window),
  save = gaction(label = "save", icon = "save",
                 handler = stub, parent = window),
  save.as = gaction(label = "save as...", icon = "save as...",
                    handler = stub, parent = window),
  quit = gaction(label = "quit", icon = "quit",
                 handler = function(...) dispose(window), parent = window),
  cut = gaction(label = "cut", icon = "cut",
                handler = stub, parent = window)
)

```

Then a toolbar list might look like this:

```

window <- gwindow("gtoolbar example")
tool_bar_list<- c(action_list[c("new","save")],
                  sep = gseparator(),
                  action_list["quit"])
tool_bar <- gtoolbar(tool_bar_list, cont = window)
gtext("Lorem ipsum ...", cont = window)

```

The `gtoolbar` constructor takes the list as its first argument. As toolbars belong to the window, the corresponding `gWidgets` objects use a `gwindow` object as the parent container. (Some of the toolkits relax this to allow

other containers.) The argument `style` can be "both", "icons", "text", or "both-horiz", to specify how the toolbar is rendered.

Menu bars and pop-up menus

Menu bars and pop-up menus are specified similarly as toolbars with menu items being defined through `gaction` instances and visual separators through `gseparator` instances. Menus differ from toolbars, as submenus require a nested structure. This is specified using a nested list as the component to describe the submenu. The lists all have named components. In this case, the corresponding name labels the submenu item. For menu bars, it is typical that all the top-level components be lists, but for pop-up menus, this wouldn't necessarily be the case.

An example of such a list might be:

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

Figure 4.11 shows this simple GUI using `gWidgetsRGtk2`. Under Mac OS X, with a native toolkit, menu bars may be drawn along the top of the screen, as is the custom of that OS.

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

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

```
no_changes <- c("save", "save.as", "cut")
keyhandler <- function(...) {
  for(i in no_changes)
    enabled(action_list[[i]]) <-
      (nchar(svalue(txt_widget)) > 0)
}
```

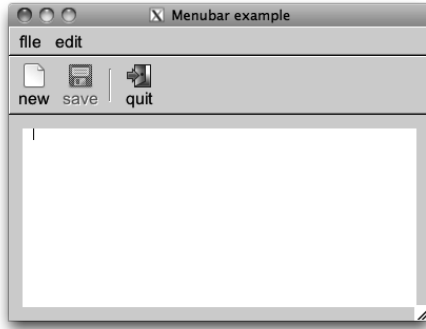


Figure 4.11: Menu bar and toolbar decorating a basic text editing widget. The “Save” icon is disabled, as there is no text typed in the buffer.

```
addHandlerKeystroke(txt_widget, handler = keyhandler)
keyhandler()
```

Pop-up menus Pop-up menus can be created for right-click events 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 pop-up, `menulist` to specify the menu, and optionally an `action` argument.

Example 4.11: Pop-up menus

This example shows how to add a simple pop-up menu to a button.

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

This page intentionally left blank

gWidgets: R-specific Widgets

The `gWidgets` package provides some R specific widgets for producing GUIs. Table 5.1 lists them.

5.1 A graphics device

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

When working with multiple devices, it becomes necessary to switch between devices. A mouse click in a `ggraphics` instance will make that device the current one. Otherwise, the `visible<-` method can be used to set the object as the current device. The `ggraphicsnotebook` creates a notebook that allows the user to easily navigate multiple graphics devices.

The default handler for the widget is set by `addHandlerClicked`. The coordinates of the mouse click, in user coordinates, are passed to the handler in the components `x` and `y`. As well, the method `addHandlerChanged`

Table 5.1: Table of constructors for R-specific widgets in `gWidgets`

Constructor	Description
<code>ggraphics</code>	Embeddable graphics device
<code>ggraphicsnotebook</code>	Notebook for multiple devices
<code>gdf</code>	Data frame editor
<code>gdfnotebook</code>	Notebook for multiple <code>gdf</code> instances
<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

is used to assign a handler to call when a region is selected by dragging the mouse. The components `x` and `y` describe the rectangle that was traced out, again in user coordinates.

This shows how the two can be used:

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

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

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

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

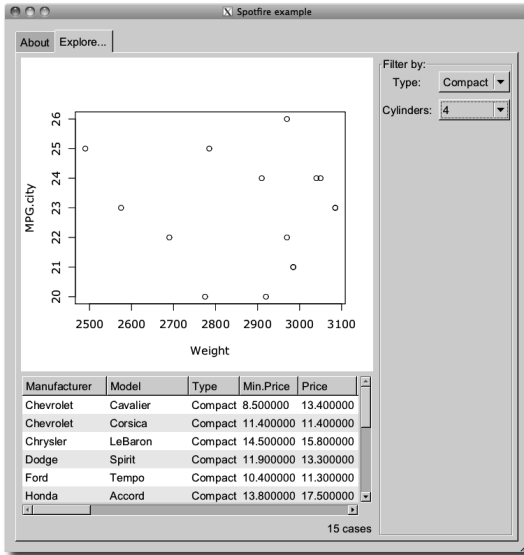


Figure 5.1: A GUI to filter a data frame and display an accompanying graphic.

```

bb <- b
tkrreplot(img)
}
}
slider <- gslider(from = 0.05, to = 2, by = 0.05, cont = group,
                  handler = f, expand = TRUE)
visible(window) <- TRUE

```

Example 5.1: A GUI for filtering and visualizing a data set

A common GUI application for data analysis consists of means to visualize, query, aggregate and filter a data set. This example shows how one can create such a GUI using `gwidgets` featuring an embedded graphics device. In addition a visual display of the filtered data, and a means to filter, or narrow, the data that is under consideration, is presented (Figure 5.1). Although, our example is not too feature rich, it illustrates a framework that can easily be extended.

This example is centered around filtering a data set; we choose a convenient one:

```
data("Cars93", package = "MASS")
```

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

```
window <- gwindow("Spotfire example", visible = FALSE)
notebook <- gnotebook(cont = window)
```

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

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

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

```
group <- ggroup(cont = notebook, label = gettext("Explore..."))
left_group <- ggroup(cont = group, horizontal = FALSE)
right_group <- ggroup(cont = group, horizontal = FALSE)
```

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

```
ggraphics(cont = left_group)
```

Our tabular 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(Cars93, cont = left_group, multiple = TRUE,
             filter.FUN = "manual")
size(tbl) <- c(500, 200) # set size
label_group <- ggroup(cont = left_group)
addSpring(label_group)
no_cases <- glabel("", cont = label_group)
```

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

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

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

```
lyt <- glayout(cont = filter_frame)
widget_list <- list() # store widgets
```

```

lyt[1,1] <- "Type:"
lyt[1,2] <- (widget_list$Type <-
             gcombobox(c("", levels(Cars93$Type)),
                       cont = lyt))
lyt[2,1] <- "Cylinders:"
lyt[2,2] <- (widget_list$Cylinders <-
             gcombobox(c("", levels(Cars93$Cyl)), cont = lyt))

```

Of course, we could use many more criteria to filter by. The above filters are naturally represented by a combo box. However, one could have used many different styles, depending on the type of data. For instance, one could employ a checkbox to filter through Boolean data, a checkbox group to allow multiple selection, a slider to pick out numeric data, or a text box to specify filtering by a string. The type of data dictates this. In this example it isn't needed, but since the layout is done, we might have code to initialize the controls in the filter. Adding such a call makes it easy to save the state of the GUI.

We now move on to the task of making the three main components – the display, the table and the filters – interact with each other. We keep this example simple, but note that if we were to extend the example we would likely write using the observer pattern introduced in Example 4.9 as that makes it easy to decouple the components of an interface. As it is, we define function calls to a) update the data frame when the filters change and b) update the graphic.

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

```

update_data_frame <- function(...) {
  vals <- lapply(widget_list, svalue)
  vals <- vals[vals != ""]
  out <- sapply(names(vals), function(i) {
    Cars93[[i]] == vals[[i]]
  })
  ind <- apply(out, 1, function(x) Reduce("&&", x))
  ## update table
  visible(tbl) <- ind
  ## update label
  nsprintf <- function(n, msg1, msg2, ...)
    ngettext(n, sprintf(msg1, n), sprintf(msg2, n), ...)
  svalue(no_cases) <- nsprintf(sum(ind), "%s case", "%s cases")
}

```

This next function is used to update the graphic. A real application would provide a more compelling plot.

```

update_graphic <- function(...) {
  ind <- visible(tbl)

```



```
if(any(ind))
  plot(MPG.city ~ Weight, data = Cars93[ind,])
else
  plot.new()
}
```

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

```
callback <- function(h, ...) {
  update_data_frame()
  update_graphic()
}
sapply(widget_list, addHandlerChanged, handler = callback)
```

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

```
addHandlerClicked(tbl, handler = function(h, ...) {
  update_graphic()
  ind <- svalue(h$obj, index = TRUE)
  points(MPG.city ~ Weight, cex = 2, col = "red", pch = 16,
         data = Cars93[ind,])
})
```

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

Finally, we draw the GUI with an initial graphic:

```
visible(window) <- TRUE
update_graphic()
```

5.2 A data frame editor

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

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

¹ For `gWidgetstcltk`, there is no native widget for editing tabular data, so the `tktable` add-on widget is used (`tktable.sourceforge.net`). A warning will be issued if this is not installed. Again, as with `gtable`, the widget under `gWidgetstcltk` is slower, but can load a moderately sized data frame in a reasonable time.

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

```

window <- gwindow("gdf example")
DF <- gdf(mtcars, cont = window)
## ... make some edits ...
new_data_frame <- DF[, ]           # store changes

```

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

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

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

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

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

5.3 Workspace browser

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

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

The default handler object calls `do.call` on the object for the function specified by name through the `action` argument. (The default is to print

a summary of the object.) This handler is called on a double click. A single click is used for selection. One can pass in other handler functions if desired.

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

Example 5.2: Using drag-and-drop with gWidgets

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

```
window <- gwindow("Drag-and-drop example")
group <- ggroup(cont = window)
workspace_browser <- gvarbrowser(cont = group)
nested_group <- ggroup(horizontal = FALSE, cont = group,
                        expand = TRUE)
ggraphics(cont = nested_group)
xlabel <- glabel("", cont = nested_group)
ylabel <- glabel("", cont = nested_group)
clear <- gbutton("clear", cont = nested_group)
```

We create a function to initialize the interface.

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

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

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

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

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

```

    plot(x, xlab = svalue(xlabel))
  } else {
    enabled(ylabel) <- TRUE
    x <- eval(parse(text = svalue(xlabel)), envir=.GlobalEnv)
    y <- eval(parse(text = svalue(ylabel)), envir=.GlobalEnv)
    plot(x, y, xlab = svalue(xlabel), ylab = svalue(ylabel))
  }
}

```

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

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

The `addDropTarget` method is used to allow a widget to receive a dropped value and to specify a handler to call when a value is dropped. The `dropdata` component of the first argument of the callback, `h`, holds the drop data. In our example below we use this to update the receiver object, either the `x` or `y` label.

```

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

```

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

5.4 Help browser

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

To add a help page, the `add` method is used, where the `value` argument describes the desired page. This can be a character string containing the

topic, a character string of the form `package:::topic` to specify the package, or a list with named components `package` and `topic`. The `dispose` method of notebooks can be used to remove the current tab.

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

5.5 Command line widget

A simple command line widget is created by the `gcommandline` constructor. This is not meant as a replacement for any of R's command lines, but is provided for light-weight usage. A text box allows users to enter 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.

5.6 Simplifying creation of dialogs

The `gWidgets` package has two means to simplify the creation of GUIs.² The `gformlayout` constructor takes a list defining a layout and produces a GUI, the `ggenericwidget` constructor can take a function name and produce a GUI based on the formal arguments of the function. This too uses a list, which can be modified by the user before the GUI is constructed. We leave the details to their manual pages.

²The `traitr` package provides another, but is not discussed here. There are similar facilities in `RGtk2Extras` for `RGtk2` and the `fgui` package can do such a thing for `tc1tk`.

Part II

The RGtk2 Package

This page intentionally left blank

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; Gdk-Pixbuf for image manipulation; GIO for synchronous and asynchronous input/output for files and network resources; 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 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 (camelCase 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 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 version of 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.

6.1 Synopsis of the RGtk2 API

Constructing a GUI with RGtk2 generally proceeds by constructing a widget and then configuring it by calling methods and setting properties. Handlers are connected to signals, and the widget is combined with other widgets to form the GUI. For example:

```
button <- gtkButton("Click Me")
button['image'] <- gtkImage(stock = "gtk-apply",
                             size = "button")
gSignalConnect(button, "clicked", function(button) {
  message("Hello World!")
})
##
window <- gtkWindow(show = FALSE)
window$add(button)
window$showAll()
```

Once one understands the syntax and themes of the above example, it is only a matter of reading through the proceeding chapters and the documentation to discover all of the widgets and their features. The rest of this chapter will explain these basic components of the API.

6.2 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-

ject class encapsulates behaviors that all instances of the class share. Every class has at most one parent through 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.

The type system supports reflection, so we can, for example, obtain a list of the ancestors for a given class:

```
gTypeGetAncestors("GtkWidget")
```

```
[1] "GtkWidget" "GObject"
```

```
[3] "GInitiallyUnowned" "GObject"
```

For those familiar with object-oriented programming in R, the returned character vector could be interpreted as if it were a class attribute on an S3 object.

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. For example, we list the interfaces implemented by GtkWidget:

```
gTypeGetInterfaces("GtkWidget")
```

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

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 Chapter 11.

6.3 Constructors

The next few sections will contribute to a unifying example that displays a button in a window. When clicked, the button will print a message to the R console. 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 GtkWidget class:



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

```
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. We prefer these shorter, more flexible constructors, such as `gtkWindow` or `gtkButton`, but note their documentation is provided by the R package author and is in addition to the formal API. These constructors can take many arguments, and only some subsets of the arguments may be specified at once. For example, this call

```
gtkImage(stock = "gtk-apply", size = "button")
```

uses only two arguments, `stock` and `size`, which must always be specified together. The entire signature is more complex:

```
args(gtkImage)
```

```
function (size, mask = NULL, pixmap = NULL, image = NULL,
          filename, pixbuf = NULL, stock.id, icon.set, animation,
          icon, show = TRUE)
```

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. A reference object will not be copied before modification. This is different from the behavior of most R objects. For example, calling `abs` on a numeric vector does not change the value assigned to the original symbol:

```
a <- -1
abs(a)
```

```
[1] 1
```

```
a
```

```
[1] -1
```

Setting the text label on our button, however, will change the original value:

```
gtkButtonSetLabel(button, "New text")
gtkButtonGetLabel(button)
```

```
[1] "New text"
```

If this widget were displayed on the screen, the label would also be updated.

The class hierarchy of an object is represented by the `class` attribute. We interpret 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"     "GObject"       "GInitiallyUnowned"
[7] "GObject"       "RGtkObject"
```

We find that the `GtkWindow` class inherits methods, properties, and signals from the `GtkBin`, `GtkContainer`, `GtkWidget`, `GObject`, `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, and methods for hiding, showing, and painting the widget. We can also query `window` for the interfaces it implements:

```
interface(window)
```

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

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.

6.4 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 relies on an understanding of how one programmatically manipulates widgets by invoking methods. In RGtk2, a method represents a type of message that is passed to a widget. Methods are implemented as functions that take an instance of their class as the first argument and instruct the widget to perform some behavior, according to any additional parameters.

Although class information is stored in the style of S3, RGtk2 introduces its own mechanism for method dispatch.¹ 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$add(widget)` RGtk2 considers `gtkBuildableAdd`, `atkImplementorInterfaceAdd`, `gtkWindowAdd`, `gtkBinAdd`, and finally finds `gtkContainerAdd`, which is called as `gtkContainerAdd(win, widget)`. The `$` 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 fewer 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("World")
button$getLabel()
```

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

Some common methods are inherited by all widgets, as they are defined in the base `GtkWidget` class. These include: `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

¹RGtk2 uses R’s standard dollar-sign notation (also used with reference classes) for class-based method dispatch.

the parent container of the widget; `modifyBg` to modify the background color of a widget; and `modifyFg` to modify the foreground color.

6.5 Properties

The GTK+ API uses properties to store public 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, if any (some properties may be fully dynamic). A list of the properties and their definitions belonging to an object is returned by its `getPropInfo` method. Calling names on the object returns the property names. 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] "use-action-appearance" "related-action"
[3] "user-data"             "name"
[5] "parent"                "width-request"
[7] "height-request"       "visible"
```

Some commonly used properties are: `parent`, to store the parent widget (if any); `user-data`, which allows us 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. The methods `get` or `set` may be used to either get or set properties of a widget, respectively. The `set` function treats the argument 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"))
window$set(icon = image[[1]], 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"
```

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

```
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.

6.6 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"))
```

```
[1] "pressed" "released" "clicked" "enter" "leave"
[6] "activate"
```

shows the “clicked” signal in addition to others. Note that this lists only 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` function adds 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 to store, but uses will be discussed later.

We demonstrate `gSignalConnect` 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(button) message("Hello World!"))
```

We now review the remaining arguments. The `data` argument allows arbitrary data to be passed to the callback. The `user.data.first` argument specifies whether the `data` argument should be the first argument to the callback or (the default) the last. The `after` argument is a logical value indicating whether the callback should be called after the default handler (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.

```

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

```

Then after clicking, we would have

```

cat(x, button$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. A return value of `TRUE` indicates that no further callbacks should be called, whereas `FALSE` indicates that the next callback should be called. In other words, the return value indicates whether the handler has consumed the event. In the following example, only the first two callbacks are executed when the user clicks the button:

```

button <- gtkButton("click")
window <- gtkWindow()
window$add(button)
gSignalConnect(button, "button-press-event",
               function(button, event, data) {
                 message("hi"); return(FALSE)
               })

```



```
gSignalConnect(button, "button-press-event",
               function(button, event, data) {
                 message("and"); return(TRUE)
               })
gSignalConnect(button, "button-press-event",
               function(button, event, data) {
                 message("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 handler, if desired, using `gSignalHandlerDisconnect`. To block a handler temporarily, call `gSignalHandlerBlock` and then `gSignalHandlerUnblock` to unblock. The help page for `gSignalConnect` gives the details.

6.7 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, such as "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
```

```
An enumeration with values:
toplevel    popup
      0      1
```

The names of the vector indicate the allowed nickname for each value of the enumeration. It is rarely necessary to use the enumeration vectors explicitly; 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
```

```
A flag enumeration with values:
  toplevel      no-window      realized
    16          32             64
  mapped        visible        sensitive
   128          256            512
parent-sensitive  can-focus      has-focus
  1024          2048            4096
  can-default    has-default    has-grab
   8192          16384           32768
  rc-style      composite-child no-reparent
  16384          131072           262144
  app-paintable receives-default double-buffered
   524288        1048576           2097152
  no-show-all
   4194304
```

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

```
window$flags()
```

```
GtkWidgetFlags: toplevel, realized, mapped, visible,
                sensitive, parent-sensitive, double-buffered
```

Flag values can be combined using `|`, the bitwise *OR*. The `&` function, the bitwise *AND*, allows us 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
```

6.8 The event loop

`RGtk2` integrates the `GTK+` and `R` event loops by treating the `R` loop as the master and iterating the `GTK+` event loop whenever `R` is idle. 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()
```

This is often useful, for example, to update a progress bar.

If we run an `RGtk2` script non-interactively, such as by assigning an icon to launch a GUI under Windows, `R` will exit after the script is finished, and the GUI will disappear just after it appears. To work around this, call the function `gtkMain` to run the main loop until the function `gtkMainQuit` is

called. Since there is no interactive session, `gtkMainQuit` should be called through some event handler.

6.9 Importing a GUI from Glade

This book focuses almost entirely on the direct programmatic construction of GUIs. Some developers prefer visually constructing a GUI by pointing, clicking and dragging in another GUI, which one might call a GUI builder, a type of RAD (Rapid Application Development) tool. Glade is the primary GUI builder for GTK+ and exports an interface as XML that is loadable by `GtkBuilder`. It is freely available for all major platforms from <http://glade.gnome.org/>. Documentation is also at that location.

We will assume that the reader has saved an interface as a `GtkBuilder` XML file named `buildable.xml` and is ready to load it with `RGtk2`:

```
builder <- gtkBuilder()
builder$addFromFile("buildable.xml")

$retval
[1] 1

$error
NULL
```

The `getObject` extracts a widget by its ID, which is specified by the user through Glade. It normally suffices to load the top-level widget, named `dialog1` in this example, and show it:

```
dialog1 <- builder$getObject("dialog1")
dialog1$showAll()
```

In order to add behaviors to the GUI, we need to register R functions as signal handlers. In Glade, the user should specify the name of an R function as a handler for some signal. `RGtk2` extends `GtkBuilder` to look up the functions and connect them to the appropriate signals. Let us assume that the user has named the `ok_button_clicked` function as the handler for the `clicked` signal on a `GtkButton`. The `connectSignals` method will establish that connection and any others in the interface:

```
ok_button_clicked <- function(button, userData) {
  message("hello world")
}
builder$connectSignals()
```

The GUI should now be ready for use.

RGtk2: Windows, Containers, and Dialogs

This chapter covers top-level windows, dialogs, and the container objects provided by GTK+.

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 the argument `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 pop-ups, 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.

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

```
[1] "Window title"
```

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

Window size The initial size of the window can be set with the `setDefaultSize` method, as shown above, 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 can be set to `FALSE`.

Adding a child component to a window A window is a container. `GtkWindow` inherits from `GtkBin`, which derives from `GtkContainer` and allows only a single child. As before, this child is added through the `add` method. We illustrate the basics by adding a simple label to a window.

```
window <- gtkWindow(show = FALSE)
window$setTitle("Hello World")
label <- gtkLabel("Hello World")
window$add(label)
```

To display multiple widgets in a window, we simply need to add a non-`GtkBin` container as the child widget. We will discuss additional container types in Section 7.2.

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. When this happens, the window manager requests that the window be deleted, and the `delete-event` signal is emitted. As with any window manager 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. For example:

```
gSignalConnect(window, "delete-event", function(event, ...) {
  dialog <- gtkMessageDialog(parent = window, flags = 0,
                             type = "question",
                             buttons = "yes-no",
                             "Are you sure you want to quit?")
  out <- dialog$run(); dialog$destroy()
  out != GtkResponseType["yes"]
})
```

(We describe the use of message dialogs in Section 7.3.) 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.

It is also possible to close a window programmatically by calling its `destroy` method:

```
window$destroy()
```

Transient windows New windows may be stand-alone 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 specifies the window with which a transient (dialog) window is associated. This hints to 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
window <- gtkWindow(show = FALSE)
window$setTitle("Top level window")
##
dialog <- gtkWindow(show = FALSE)
dialog$setTitle("dialog window")
dialog$setTransientFor(window)
dialog$setPosition("center-on-parent")
dialog$setDestroyWithParent(TRUE)
window$show()
dialog$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 in Section 7.3, 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 as a fixed 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 can ill afford to manage a dynamic layout explicitly. Thus, GTK+ implements automatic layout in the form of container widgets.

Basics

In GTK+, the widget hierarchy is built when children are added to a parent container. In this example, a window is made the parent of a label:

```
window <- gtkWindow(show=FALSE)
window$setTitle("Hello World")
```

```
label <- gtkLabel("Hello World")
window$add(label)
```

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 we could do:

```
window$getChild()[ 'label' ]
```

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

The `[[` method accesses the child widgets by number, as a convenient wrapper around the `getChildren` method:

```
window[[1]][ 'label' ]
```

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

Conversely, 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. For instance

```
window$remove(label)
window$add(label)
```

To remove a widget from the screen but not its container, use the `hide` method on the widget. The `reparent` method is a convenience for moving a widget between containers that ensures the child is not garbage collected during the transition.¹

Widget size negotiation

We have already seen perhaps the simplest automatic layout container, `GtkBin`, which fills all of its space with its child. Despite the apparent simplicity, 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 then consumes the allocated space. Consider the previous example of adding a label to a window:

```
window <- gtkWindow()
window$setTitle("Hello World")
label <- gtkLabel("Hello World")
window$add(label)
```

¹An object becomes available for garbage collection when it has no references to it, which can happen if it is removed from the parent container.

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.

```
label$getAllocation()$allocation
```

x	y	width	height
0	0	200	200

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:

```
window <- gtkWindow(show = FALSE)
window$setTitle("Hello World")
label <- gtkLabel("Hello World")
window$add(label)
window$show()
label$getAllocation()$allocation
```

x	y	width	height
0	0	83	18

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 nontrivial 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 multi-child container in GTK+ is the box (implemented in class `GtkBox`), which packs its children as if they were in a box. Instances of `GtkBox` are constructed by `gtkHBox` and `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 and align themselves in both dimensions, 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`.

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 five pixels of space between children. 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 against the left and right side (top and bottom for a `GtkVBox`), respectively. For this explanation, we restrict ourselves to `packStart`, since `packEnd` works the same except for the direction. Below, we pack two buttons, `button_a` and `button_b` against the left side:

```
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 natural size 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 and the widget is aligned in the center of its space. When a widget is packed with the `fill` parameter set to `TRUE`, the widget is resized 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 a heterogeneously spaced `GtkHBox`, we pass `FALSE` as the first argument to the constructor, as in the following code:

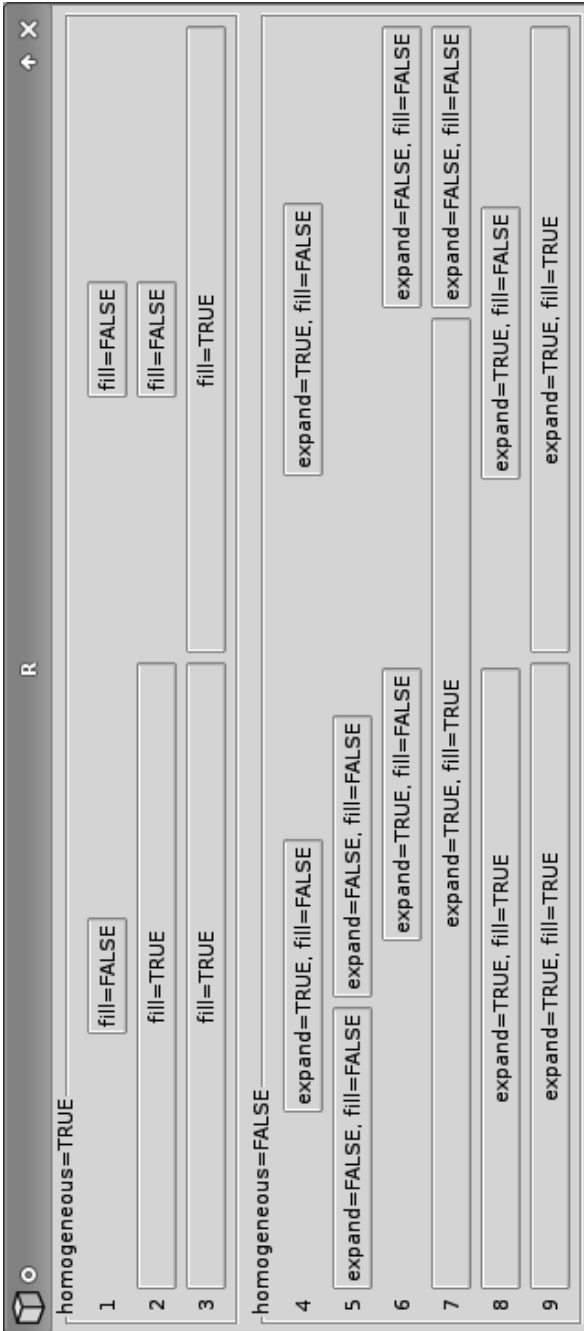


Figure 7.1: A screenshot demonstrating the effect of packing two buttons into `GtkHBox` instances using the `packStart` method with different combinations of the `expand` and `fill` settings. The effect of the `homogeneous` spacing setting on the `GtkHBox` is also shown.

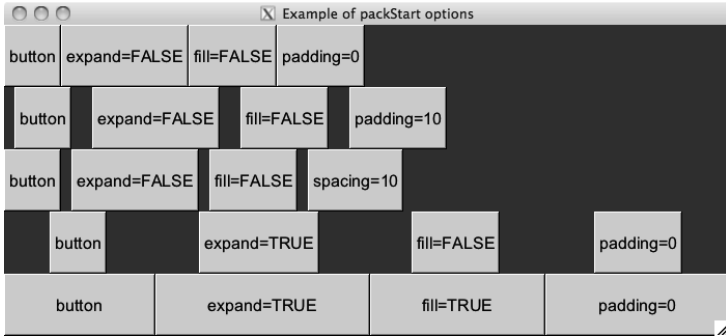


Figure 7.2: Examples of packing widgets into a box container. The top row shows no padding, whereas the second and third illustrate the difference between padding (an amount around each child) and spacing (an amount between the children). The last two rows show the effect of fill when `expand=TRUE`.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

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

A heterogeneous layout is freed of the restriction that all widgets must be given the same amount of available space; it needs only 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.

Padding There are several ways to add space around widgets in a box container. The `spacing` argument for the constructors specifies the amount of space between the cells, in pixels. 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 8.3 provides an example and Figure 7.2 an illustration.

Positioning The `reorderChild` method reorders the child widgets. The new position of the child is specified using 0-based indexing. This code will move the third child of `hbox` to the second position:

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

Alignment

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

```
window <- gtkWindow(); window$setTitle("Hello World")
label <- gtkLabel("Hello World")
window$add(label)
```

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 so 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:

```
label["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:

```
window <- gtkWindow(); window$setTitle("Hello World")
alignment <- gtkAlignment()
alignment$set(xalign = 0, yalign = 0.5, xscale = 0, yscale=1)
window$add(alignment)
```

```
label <- gtkLabel("Hello World")
alignment$add(label)
```

7.3 Dialogs

GTK+ provides a number of convenient dialogs for the most common use cases, as well as a general infrastructure for constructing custom dialogs. A dialog is a window that generally consists of an icon, a content area, and an action area containing a row of buttons representing the possible user responses. Typically, a dialog belongs to a main application window and might be modal, in which case input is blocked to other parts of the GUI. `GtkDialog` represents a generic dialog and serves as the base class for all special-purpose dialogs in GTK+.

Message dialogs

Communicating textual messages to the user is perhaps the most common application of a dialog. GTK+ provides the `gtkMessageDialog` convenience wrapper for `GtkDialog` for creating a message dialog showing a primary and secondary message. We construct one presently:

```
window <- gtkWindow(); window['title'] <- "Parent window"
#
dialog <- gtkMessageDialog(parent=window,
                           flags="destroy-with-parent",
                           type="question",
                           buttons="ok",
                           "My message")
dialog['secondary-text'] <- "A secondary message"
```

The `flags` argument allows us to specify a combination of values from `GtkDialogFlags`. These include `destroy-with-parent` and `modal`. Here, the dialog will be destroyed upon destruction of the parent window. The `type` argument specifies the message type, using one of the four values from `GtkMessageType`, which determines the icon that is placed adjacent to the message text. The `buttons` argument indicates the set of response buttons with a value from `GtkButtonsType`. The remaining arguments are pasted together into the primary message. The dialog has a `secondary-text` property that can be set to give a secondary message.

Dialogs are optionally modal. Below, we enable modality by calling the `run` method, which will additionally block the R session:

```
response <- dialog$run()
if(response == GtkResponseType["cancel"] ||
    response == GtkResponseType["close"] ||
    response == GtkResponseType["delete-event"]) {
```

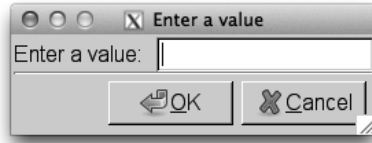


Figure 7.3: Using `gtkDialog` to create a custom dialog, in this case one showing a label and entry widget.

```
## pass
} else if(response == GtkResponseType["ok"]) {
  message("Ok")
}
dialog$destroy()
```

The return value can then be inspected for the action, such as which button was pressed. `GtkMessageDialog` will return response codes from the `GtkResponseType` enumeration. We will see an example of asynchronous response handling in the next section.

Custom dialogs

The `gtkDialog` constructor returns a generic dialog object which can be customized, in terms of its content and response buttons. Usually, a `GtkDialog` is constructed with `gtkDialogNewWithButtons`, as a dialog almost always contains a set of response buttons, such as Ok, Yes, No and Cancel. In this example, we will create a simple dialog showing a label and text entry:

```
dialog <- gtkDialogNewWithButtons(title = "Enter a value",
  parent = NULL, flags = 0,
  "gtk-ok", GtkResponseType["ok"],
  "gtk-cancel", GtkResponseType["cancel"],
  show = FALSE)
```

Buttons are added with a label and a response ID, and their order is taken from their order in the call. There is no automatic ordering based on an operating system's conventions. When the button label matches a stock ID, the icon and text are taken from the stock definition. We used standard responses from `GtkResponseType`, although in general the codes are simply integer values; interpretation is up to the programmer.

The dialog has a content area; an instance of `GtkVBox`. To complete our dialog, we place a labeled text entry into the content area:

```
hbox <- gtkHBox()
hbox['spacing'] <- 10
```

```
#
hbox$packStart(gtkLabel("Enter a value:"))
entry <- gtkEntry()
hbox$packStart(entry)
#
vbox <- dialog$getContentArea()
vbox$packStart(hbox)
```

The content is placed above the button box, with a separator between them.

In the message dialog example, we called the `run` method to make the dialog modal. To make a non-modal dialog, do not call `run` but connect to the response signal of the modal dialog. The response code of the clicked button is passed to the callback:

```
gSignalConnect(dialog, "response",
               f=function(dialog, response, user.data) {
                 if(response == GtkResponseType["ok"])
                   print(entry$getText()) # Replace this
                 dialog$Destroy()
               })
dialog$showAll()
dialog$setModal(TRUE)
```

File chooser

A common task in a GUI is the selection of files and directories, for example to load or save a document. `GtkFileChooser` is an interface shared by widgets that choose files. GTK+ provides three such widgets. The first is `GtkFileChooserWidget`, which can be placed anywhere in a GUI. The other two are based on the first. `GtkFileChooserDialog` embeds the chooser widget in a modal dialog, while `GtkFileChooserButton` is a button that displays a file path and launches the dialog when clicked.

Example 7.1: An open-file dialog

Here, we demonstrate most commonly used of the three file-choosing dialog. An open file dialog can be created with:

```
dialog <- gtkFileChooserDialog(title = "Open a file",
                              parent = NULL, action = "open",
                              "gtk-ok", GtkResponseType["ok"],
                              "gtk-cancel", GtkResponseType["cancel"],
                              show = FALSE)
```

The dialog constructor allows us to specify a title, a parent, and an action,; either `open`, `save`, `select-folder`, or `create-folder`. In addition, the dialog buttons must be specified, as with the last example, using `gtkDialogNewWithButtons`.

We connect to the response signal

```
gSignalConnect(dialog, "response",
  f = function(dialog, response, data) {
    if(response == GtkResponseType["ok"]) {
      filename <- dialog$getFilename()
      print(filename)
    }
    dialog$destroy()
  })
```

The file selected is returned by `getFilename`. If multiple selection is enabled (via the `select-multiple` property) we should call the plural `getFilenames`.

For the open dialog, we may wish to specify one or more filters that narrow the available files for selection:

```
fileFilter <- gtkFileFilter()
fileFilter$setName("R files")
fileFilter$addPattern("*.R")
fileFilter$addPattern("*.Rdata")
dialog$addFilter(fileFilter)
```

The `gtkFileFilter` function constructs a filter, which is given a name and a set of file-name patterns, before being added to the file chooser. Filtering by MIME type is also supported.

The save file dialog would be similar. The initial file name could be specified with `setFilename`, or folder with `setFolder`. The `do-overwrite-confirmation` property controls whether the user is prompted when attempting to overwrite an existing file.

Other features not discussed here include embedding of preview and other custom widgets, and specifying shortcut folders.

Other choosers

There are several other types of dialogs for making common types of selections. These include `GtkCalendar` for picking dates, `GtkColorSelectionDialog` for choosing colors, and `GtkFontSelectionDialog` for fonts. These are very high-level dialogs that are trivial to construct and manipulate, at a cost of flexibility.

Print dialog

Rendering documents for printing is outside our scope; however, we will mention that `GtkPrintOperation` can launch the native, platform-specific print dialog for customizing a printing operation. See Example 8.11 for an example of printing R graphics using `cairoDevice`.

7.4 Special-purpose containers

In Section 7.2, 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 many methods.

Framed containers

The `gtkFrame` function constructs a container that draws a decorative, labeled frame around its single child:

```
frame <- gtkFrame("Options")
vbox <- gtkVBox()
vbox$packStart(gtkCheckButton("Option 1"), FALSE)
vbox$packStart(gtkCheckButton("Option 2"), FALSE)
frame$add(vbox)
```

A frame is useful for visually segregating a set of conceptually related widgets from the rest of the GUI. The type of decorative shadow is stored in the `shadow-type` property. The `setLabelAlign` aligns the label relative to the frame. This is to the left, by default.

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`:

```
expander <- gtkExpander("Advanced")
expander$add(frame)
```

Use `gtkExpanderNewWithMnemonic` if a mnemonic is desired. 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.

We create a notebook and add some pages:

```
notebook <- gtkNotebook()
notebook$appendPage(gtkLabel("Page 1"), gtkLabel("Tab 1"))
```

```
[1] 0
```

```
notebook$appendPage(gtkLabel("Page 2"), gtkLabel("Tab 2"))
```

```
[1] 1
```

A page specification consists of a widget for the page and a widget for the tab. Any type of widget is accepted, although a label is typically used for the tab. This flexibility allows for more complicated tabs, such as a box container with a label and a close icon.

The tabs can be positioned on any of the four sides of the notebook; this depends on the `tab-pos` property, with a value from `GtkPositionType`: "left", "right", "top", or "bottom". By default, the tabs are on top. We move the current ones to the bottom:

```
notebook['tab-pos'] <- "bottom"
```

Methods and properties that affect pages expect the page index, instead of the page widget. To map from the child widget to the page number, use the method `pageNum`. The `page` property holds the zero-based index of the active tab. We make the second tab active:

```
notebook['page'] <- 1
notebook['page']
```

```
[1] 1
```

To move sequentially through the pages, call the methods `nextPage` and `prevPage`. The signal `switch-page` is emitted when the current page changes.

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, as long as they belong to the same group, which depends on the `group-id` property. 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 pop-up menu listing all of the tabs for direct navigation.

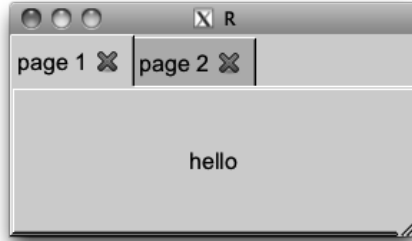


Figure 7.4: Simple illustration of customized tab in a notebook. These include close buttons.

Example 7.2: 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) {  
    icon <- gtkImage(pixbuf =  
      object$renderIcon("gtk-close", "button", size = "menu"))  
    closeButton <- gtkButton()  
    closeButton$setImage(icon)  
    closeButton$setRelief("none")  
    ##  
    label <- gtkHBox()  
    label$packStart(gtkLabel(label.text))  
    label$packEnd(closeButton)  
    ##  
    gSignalConnect(closeButton, "clicked", function(button) {  
      index <- object$pageNum(child)  
      object$removePage(index)  
    })  
    object$insertPage(child, label, position)  
  }
```

Here is a simple demonstration of its usage:

```
window <- gtkWindow()  
notebook <- gtkNotebook(); window$add(notebook)  
notebook$insertPageWithCloseButton(gtkButton("hello"),  
                                   label.text = "page 1")  
notebook$insertPageWithCloseButton(gtkButton("world"),  
                                   label.text = "page 2")
```

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 scroll bars 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. Scrolled windows are most often used with potentially large widgets like table views and when displaying images and graphics.

Our example will embed an R-graphics device in a scrolled window and allow the user to zoom in and out and pull on the scroll bars to pan the view. First, we create an R-graphics device using the `cairoDevice` package

```
library(cairoDevice)
device <- gtkDrawingArea()
device$setSizeRequest(600, 400)
asCairoDevice(device)
```

and then embed it within a scrolled window:

```
scrolled <- gtkScrolledWindow()
scrolled$addWithViewport(device)
```

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, like our `GtkDrawingArea`, must be embedded within the proxy `GtkViewport`. The `GtkScrolledWindow` convenience method `addWithViewport` allows the programmer to skip the `GtkViewport` step.

Next, we define a function for scaling the plot:

```
zoomPlot <- function(x = 2.0) {
  allocation <- device$getAllocation()$allocation
  device$setSizeRequest(allocation$width * x,
                        allocation$height * x)
  updateAdjustment <- function(adjustment) {
    adjustment$setValue(x * adjustment$getValue() +
                        (x - 1) * adjustment$getPageSize()/2)
  }
  updateAdjustment(scrolled$getHadjustment())
  updateAdjustment(scrolled$getVadjustment())
}
```

The function gets the current size allocation from the device, scales it by x , and requests the new size. It then scrolls the window to preserve the center point. The state of each scroll bar is represented by a `GtkAdjustment`. We update the value of the horizontal and vertical adjustments to scroll the

window. The value of an adjustment corresponds to the left/top position of the window, so we adjust by half the page size after scaling the value.

We had key-press events, so that pressing + zooms in and pressing - zooms out:

```
gSignalConnect(scrolled, "key-press-event",
               function(scrolled, event) {
                 key <- event[["keyval"]]
                 if (key == GDK_plus)
                   zoomPlot(2.0)
                 else if (key == GDK_minus)
                   zoomPlot(0.5)
                 TRUE
               })
```

Despite its name, the scrolled window is not a top-level window. Thus, it needs to be added to a top-level window:

```
win <- gtkWindow(show = FALSE)
win$add(scrolled)
win$showAll()
```

Finally, a basic scatterplot is displayed in the viewer:

```
plot(mpg ~ hp, data = mtcars)
```

The properties `hscrollbar-policy` and `vscrollbar-policy` determine when the scroll bars are drawn. By default, they are always drawn. The "automatic" value from the `GtkPolicyType` enumeration draws the scroll bars only if needed, i.e, if the child widget requests more space than can be allocated. The `setPolicy` method allows both to be set at once.

Divided containers

The `gtkHPaned` and `gtkVPaned` constructors create containers that hold two child widgets, arranged horizontally or vertically and separated by a divider displaying a handle allowing the user to adjust the allocation of space between the child components. We will demonstrate only the horizontal pane `GtkHPaned` here, without loss of generality.

First, we construct an instance of `GtkHPaned`:

```
paned <- gtkHPaned()
```

The two children can be added two different ways. The simplest approach calls `add1` and `add2` for adding the first and second child, respectively.

```
paned$add1(gtkLabel("Left (1)"))
paned$add2(gtkLabel("Right (2)"))
```

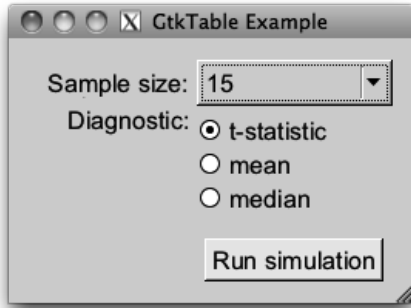


Figure 7.5: A basic dialog using a `gtkTable` container for layout.

This configures the container such that both children are allowed to shrink and only the second widget can expand. Such a configuration is appropriate for a GUI with main widget and a side pane to the left. More flexibility is afforded by the methods `pack1` and `pack2`, which have arguments for specifying whether the child should expand ("resize") and/or "shrink". Here we add the children such that both can expand and shrink:

```
paned$pack1(gtkLabel("Left (1)"), resize = TRUE, shrink=TRUE)
paned$pack2(gtkLabel("Right (2)"), resize = TRUE, shrink=TRUE)
```

After children are added, they can be retrieved 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` are useful for converting a percentage into a screen position. The `move-handle` signal is emitted when the handle position is changed.

Tabular layout

`GtkTable` is a container for laying out objects in a tabular (or grid) format. It is *not* meant for displaying tabular data. The container divides its space into cells of a grid, and a child widget may occupy one or more cells. The allocation of space within a row or column follows logic similar to that of box layouts. The most common use case of a `GtkTable` is a form layout, which we will demonstrate in our example.

Example 7.3: Dialog layout

This example shows how to lay out a form in a dialog with some attention paid to how the widgets are aligned and how they respond to resizing of the window.

Our form layout will require three rows and two columns:

```
table <- gtkTable(rows = 3, columns = 2, homogeneous = FALSE)
```

By default, the cells are allowed to have different sizes. This may be overridden by passing "homogeneous = TRUE" to the constructor, which forces all cells to have the same size.

We construct the widgets that will be placed in the form:

```
size_label <- gtkLabel("Sample size:")
size_combo <- gtkComboBoxNewText()
sapply(c(5, 10, 15, 30), size_combo$appendText)
##
diag_label <- gtkLabel("Diagnostic:")
diag_radio <- gtkVBox()
radiogp <- list()
radiogp$t <- gtkRadioButton(label = "t-statistic")
radiogp$mean <- gtkRadioButton(radiogp, label = "mean")
radiogp$median <- gtkRadioButton(radiogp, label = "median")
sapply(radiogp, diag_radio$packStart)
##
submit_vbox <- gtkVBox()
submit_vbox$packEnd(gtkButton("Run simulation"), expand=FALSE)
```

We align the labels to the right, up against their corresponding entry widgets, which are left-aligned:

```
size_label['xalign'] <- 1
diag_label['xalign'] <- 1; diag_label['yalign'] <- 0
diag_align <- gtkAlignment(xalign = 0)
diag_align$add(diag_radio)
```

The labels are aligned through the `GtkMisc` functionality inherited by `GtkLabel`. The `GtkVBox` with the radio buttons does not support this, so we have embedded it within a `GtkAlignment` instance. We have aligned the diagnostic label to the top of its cell; otherwise, it would have been centered vertically. The radio buttons are left-aligned, up against the label (cf. Figure 7.5).

Child widgets are added to a `GtkTable` instance through its `attach` method. The child 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 can be overridden by specifying additional arguments to `attach`.

The following attaches the combo box, radio buttons, and their labels to the table:

```

table$attach(size_label, left.attach = 0,1, top.attach = 0,1,
             xoptions = c("expand", "fill"), yoptions = "")
table$attach(size_combo, left.attach = 1,2, top.attach = 0,1,
             xoptions = "fill", yoptions = "")
##
table$attach(diag_label, left.attach = 0,1, top.attach = 1,2,
             xoptions = c("expand", "fill"),
             yoptions = c("expand", "fill"))
##
table$attach(diag_align, left.attach = 1,2, top.attach = 1,2,
             xoptions = c("expand", "fill"), yoptions = "")
##
table$attach(submit_vbox, left.attach = 1,2, top.attach = 2,3,
             xoptions = "", yoptions = c("expand", "fill"))

```

The labels are allowed to expand and fill in the x direction, because correct alignment, to the right, requires them to have the same size. The combo box is instructed to fill its space, as it would otherwise be undesirably small, due to its short menu items.

We can add spacing to the right of cells in a particular row or column. Here we add five pixels of space to the right of the label column:

```
table$setColSpacing(0, 5)
```

We complete the example by placing the table into a window:

```

window <- gtkWindow(show=FALSE)
window['border-width'] <- 14
window$setTitle("GtkTable Example")
window$add(table)

```


This page intentionally left blank

RGtk2: Basic Components

In this chapter we cover many of the basic controls of GTK+.

8.1 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 8.1: Button constructors

```
window <- gtkWindow(show = FALSE)
window$setTitle("Various buttons")
window$setDefaultSize(400, 25)
hbox <- gtkHBox(homogeneous = FALSE, spacing = 5)
window$add(hbox)
button <- gtkButtonNew()
button$setLabel("long way")
hbox$packStart(button)
hbox$packStart(gtkButton(label = "label only") )
hbox$packStart(gtkButton(stock.id = "gtk-ok") )
hbox$packStart(gtkButtonNewWithMnemonic("_Mnemonic") )
window$show()
```



Figure 8.1: Various buttons.

A `GtkButton` is simply a clickable region on the screen that is rendered 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.

Passing the `stock.id` argument to `gtkButton` will use decorations associated with a so-called stock identifier (see Section 8.2). For example, “`gtk-ok`” would produce a button with a theme-dependent image (such as a check mark) and the “Ok” label, with the appropriate mnemonic (see below) and language translation. The available stock identifiers are listed by `gtkStockListIds`.

The `gtkButtonNewWithMnemonic` constructor creates a button with a mnemonic. A mnemonic is a key press that will activate the button and is indicated by prefixing the character with an underscore. In our example, we pass the string “`_Mnemonic`”, so pressing `Alt-M` will effectively press the button.

Signals The `clicked` signal is emitted when the button is clicked with the mouse, when the associated mnemonic is pressed, 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 8.2: Callback example for `gtkButton`

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

As buttons are intended to call an action immediately after being clicked, it is advisable to make them insensitive to user input when the action is not possible. For example, we set our button to be insensitive through:

```
button$setSensitive(FALSE)
```

Windows often have a default action. For example, if a window contains a form, the default action submits the form. If a button executes the

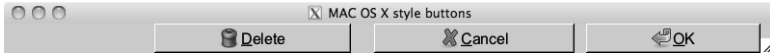


Figure 8.2: Example using stock buttons with extra spacing added between the delete and cancel buttons.

default action for the window, the button 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.) The `GtkDialog` widget and its derivatives facilitate the use of buttons in this manner (see Section 7.3).

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 10.5.

Example 8.3: Spacing between buttons

This example shows how to pack buttons into a box so that the spacing between the similar buttons is twelve pixels, while potentially dangerous buttons are separated from the rest by twenty-four pixels, as per the Apple human interface guidelines.

GTK+ provides the widget `GtkHButtonBox` for organizing buttons in a manner consistent across an application. However, the default layout modes would not yield the desired spacing. As such, we will illustrate how to customize the spacing. We assume that our parent container, `hbox`, is a horizontal box container.

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

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

We specify the padding as we pack the widgets into the box, from right to left, with `packEnd`:

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

The padding occurs to the left and right of the child. The `ok` button is given no padding. The `cancel` button is packed with twelve pixels of spacing, which separates it from the `ok` button. Recognizing the `delete` button as potentially irreversible, we add twelve pixels of separation between it and the `cancel` button, for a total of twenty-four pixels. The blank label pushes

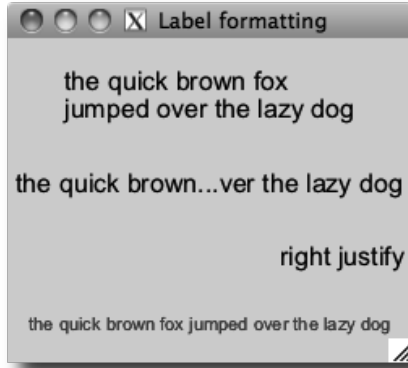


Figure 8.3: Various formatting for a label: wrapping, alignment, “ellipsizing,” and Pango markup.

the buttons against the right side of the box. In the last line, we instruct the ok button to grab focus, so that it becomes the default button:.

8.2 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, which takes the label text as its first argument. This text can be set with either `setLabel` or `setText` and retrieved with either `getLabel` or `getText`. The difference between the two is that the former respects formatting marks.

Example 8.4: Label formatting

As most text in a GTK+ GUI is ultimately displayed by `GtkLabel`, there are many formatting options available. This example demonstrates a sample of these (Figure 8.3).

```
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) # no. characters
## Set ellipsis to shorten long text
ellipsized <- gtkLabel(string)
ellipsized$setEllipsize("middle")
## Right justify text lines
```

```

## use xalign property for aligning entire block
rightJustified <- gtkLabel("right justify")
rightJustified$setJustify("right")
rightJustified['xalign'] <- 1
## PANGO markup
pangoLabel <- gtkLabel()
tmpl <- "<span foreground='blue' size='x-small'>%s</span>"
pangoLabel$setMarkup(sprintf(tmpl, string))
#
sapply(list(basicLabel,ellipsized,rightJustified, pangoLabel),
       vbox$packStart, expand = TRUE, fill = TRUE)
window$showAll()

```

Many of the text formatting options are demonstrated in Example 8.4. Line wrapping is enabled with `setLineWrap`. Labels also support explicit line breaks, specified with “\n.” The `setWidthChars` 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 some of their text replaced 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.

Pango markup 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. Hyperlinks are possible with `<a>`, as of version 2.18, and similar logic to `browseURL` is implemented for launching a web browser. Connect to the `activate_link` signal to override it. More complicated markup involves the `` tag markup, such as `some text`. As with HTML, the text 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.

For efficiency reasons `GtkLabel` does not receive any input events. It lacks an underlying `GdkWindow`, meaning that there are no window-system resources allocated for receiving the events. Thus, to make a label inter-

active, we must first embed it within a `GtkEventBox`, which provides the `GdkWindow`.

Images

It is often said that a picture can be worth a thousand words. Applying this to a GUI, good images can be worth thousands of screen pixels, as they can compactly represent ideas and actions. `GtkImage` is the widget that displays images. The constructor `gtkImage` creates images from various in-memory image representations, files, and other sources. Images can be loaded after construction, as well. For example, the `setFromFile` method loads an image from a file.

Example 8.5: 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.

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

The size of the image is taken as the size allocated to the box `hbox`. 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 <- hbox$getAllocation()$allocation
width <- theSize$width; height <- theSize$height
```

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

```
require(cairoDevice)
pixmap <- gdkPixmap(drawable = NULL,
                    width = width, height = height, depth=24)
asCairoDevice(pixmap)
hist(rnorm(100))
```

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

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

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 we wish to connect to such events.

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 is returned, as a list, by `gtkStockListIds`. The first three are:

```
head(unlist(gtkStockListIds()), n=3)
```

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

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 a stock identifier.

8.3 Input controls

Text entry

The widgets explained thus far are largely static, i.e., it is not possible to edit a label or image. GTK+ has two different widgets for editing text. One is optimized for multiline text documents, the other for single-line entry. We will discuss complex multiline text editing in Section 9.6. For entering a single line of text, the `GtkEntry` widget is appropriate:

```
entry <- gtkEntry()
```

The `text` property stores the text. This can be set with the method `setText` and retrieved with `getText`. When the user has committed an entry, e.g., by pressing the enter key, the `activate` signal is emitted. Here we connect to this signal to obtain the entered text upon activation:

```
gSignalConnect(entry, "activate", function() {
  message("Text entered: ", entry$getText())
})
```




Figure 8.4: Illustration of adding an icon to a `GtkEntry` instance to indicate whether the text entered is valid.

Sometimes the length of the text needs to be constrained to some number of characters. The `max` argument to `gtkEntry` specifies this, but that usage is deprecated, in favor of the `setMaxLength` method.

The `GtkEditable` interface Editing text programmatically relies on the `GtkEditable` interface, which `GtkEntry` implements. The method `insertText` inserts text before a position specified by a 0-based index. The return value is a list with the component position indicating the position *after* the new text. The `deleteText` method deletes text between two positions.

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`.

Advanced `GtkEntry` features `GtkEntry` has a number of features beyond basic text entry, including: completion, buffer sharing, icons, and progress reporting. We discuss completion in Section 9.4 and shared buffers in Section 9.5. The progress reporting API, introduced with version 2.16, is virtually identical to that of `GtkProgressBar`, introduced in Section 8.4. We treat icons here. This feature has been present since version 2.16.

We can set an icon on an entry from a `GdkPixbuf`, stock ID, icon name, or `GIcon` (Figure 8.4). Two icons are possible, one at the beginning (primary) and one at the end (secondary). A common use would be to place a search icon in an entry widget, were it used for searching. In our example below, an entry might listen to its input and update its icon to indicate whether the entered text is valid (in this case, consisting only of letters):

```
validatedEntry <- gtkEntry()
gSignalConnect(validatedEntry, "changed", function(entry) {
  text <- entry$getText()
  if (nzchar(gsub("[a-zA-Z]", "", text))) {
    entry$setIconFromStock("primary", "gtk-no")
    entry$setIconTooltipText("primary",
```

```

                                "Only letters are allowed")
  } else {
    entry$setIconFromStock("primary", "gtk-yes")
    entry$setIconTooltipText("primary", NULL)
  }
})
validatedEntry$setIconFromStock("primary", "gtk-yes")

```

We add a tooltip on the error icon to indicate the nature of the problem to the user. Icons can also be made clickable and used as a source for drag-and-drop operations.

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 one aspect of the application state. The simplest type of state variable is binary (Boolean) and is usually proxied by a `GtkCheckButton`.

A `GtkCheckButton` is constructed by `gtkCheckButton`:

```
checkButton <- gtkCheckButton("Option")
```

The state of the binary variable is represented by the `active` property. We check our button:

```
checkButton['active']
```

```
[1] FALSE
```

```
checkButton['active'] <- TRUE
```

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:

```
gSignalConnect(checkButton, "toggled", function(button) {
  state <- ifelse(button$active, "active", "inactive")
  message("Button is ", state)
})
```

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 is drawn as depressed while the state variable is `TRUE`, instead of relying on a checkbox to communicate the binary value.

Radio-button groups

GTK+ provides two widgets 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, and only one button in a group may be active at once.

Example 8.6: Basic radio-button usage

When we construct a radio button, we need to add it 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 the group list is passed to the constructor, the newly created button is added to the group:

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

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 supply idiom that avoids the need to allocate a list and populate it in a for loop.

We add each button to a vertical box:

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

We can set and query which button is active:

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

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

The toggle signal is emitted when a button is toggled. We need to connect a handler to each button:

```
sapply(radiogp, gSignalConnect, "toggled",      # connect each
  f = function(button, data) {
    if(button['active']) # set before callback
      message("clicked", button$getLabel(), "\n")
  })
```

Example 8.7: Radio group via a FromWidget constructor

In this example, we illustrate using the `gtkRadioButtonNewWithLabelFromWidget` function to add new buttons to the group:

```
radiogp <- gtkRadioButton(label=labels[1])
btns <- sapply(labels[-1], gtkRadioButtonNewWithLabelFromWidget,
              group = radiogp)
window <- gtkWindow()
window['title'] <- "Radio group example"
vbox <- gtkVBox(); window$add(vbox)
sapply(rev(radiogp$getGroup()), gtkBoxPackStart, object = vbox)
```

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. Thus, we need to call `rev` to reverse the list before packing the widgets into the box.

Combo boxes

The combo box is a more space-efficient alternative to a radio button group and is better suited when there are a large number of options. A basic, text-only `GtkComboBox` is constructed by `gtkComboBoxNewText`. In Section 9.3 we will discuss combo boxes that are based on an external data model.

We can construct and populate a simple combo box with:

```
combo <- gtkComboBoxNewText()
sapply(c("two.sided", "less", "greater"), combo$appendText)
```

The index of the currently active item is stored in the `active` property. The index, as usual, is 0-based, and a value of `-1` indicates that no value is selected (the default):

```
combo['active']
```

```
[1] -1
```

The `getActiveText` method retrieves the text shown by the basic combo box.

When the active index changes, the `changed` signal is emitted. The handler then needs to retrieve the active index:

```
gSignalConnect(combo, "changed",
              f = function(button, ...) {
                if(button$getActive() < 0)
                  message("No value selected")
                else
                  message("Value is", button$getActiveText())
              })
```

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

Example 8.8: Using one combo box to populate another

The goal of this example is to populate a combo box of variables whenever a data frame is selected in another. We use two convenience functions from the `ProgGUIinR` package to find the possible data frames, and for a data frame to find its variables.

We create the two combo boxes and the enclosing window:

```
window <- gtkWindow(show = FALSE)
window$setTitle("gtkComboBox example")
df_combo <- gtkComboBoxNewText()
var_combo <- gtkComboBoxNewText()
```

Our layout uses boxes. To add a twist, we will hide our variable combo box until after a data frame has been initially selected.

```
vbox <- gtkVBox(); window$add(vbox)
#
vbox1 <- gtkHBox(); vbox$packStart(vbox1)
vbox1$packStart(gtkLabel("Data frames:"))
vbox1$packStart(df_combo)
#
vbox2 <- gtkHBox(); vbox$packStart(vbox2)
vbox2$packStart(gtkLabel("Variable:"))
vbox2$packStart(var_combo)
vbox2$hide()
```

Finally, we configure the combo boxes. When a data frame is selected, we first clear out the variable combo box and then populate it:

```
sapply(avail_dfs(), df_combo$appendText)
df_combo$setActive(-1)
#
gSignalConnect(df_combo, "changed", function(df_combo, ...) {
  var_combo$getModel()$clear()
  sapply(find_vars(df_combo$getActiveText()),
         var_combo$appendText)
  vbox2$show()
})
```

An extension of `GtkComboBox`, `GtkComboBoxEntry`, replaces the main button with a text entry. This supports the entry of arbitrary values, in addition to those present in the menu.

Sliders and spin buttons

The slider widget and spin-button widget allow selection from a regularly spaced, semi-continuous list of values. Both have their possible values for selection determined by an instance of `GtkAdjustment`, which is used to represent ranges that have an upper and lower bound with step and page increments. This adjustment may be specified to the constructor or, more frequently, will be created by the widget after an appropriate specification of the range.

Sliders Sliders are implemented by `GtkScale` with constructors `gtkHScale` and `gtkVScale`, the difference being the orientation.

These constructors have arguments `min`, `max`, and `step` to specify the range, if an adjustment is not specified.

The `value` property stores the currently selected value. When this is changed, the `value-changed` signal is emitted.

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`.

In Example 8.12 we show how a slider can be used to update a 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 through the `min`, `max`, and `step` arguments. The argument `digits` is used to configure how many digits are displayed, and `climb.rate` can adjust how fast the display changes when the button is held depressed.

As with `GtkScale`, the `value` property holds the state and the `value-changed` signal is emitted when this changes.

A spin button has a few additional features. 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 whether the sequence will “wrap” around at the bounds.

Example 8.9: A range widget

This example shows how to make a range widget that combines both the slider and spin button to choose a single number (Figure 8.5). Such a widget is useful, 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.

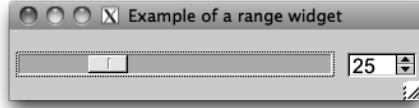


Figure 8.5: A range widget with coordinated slider and spin box sharing the same `GtkAdjustment` instance.

We name our scale parameters according to the corresponding arguments to the `seq` function:

```
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 then 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, but not the spin button, set to expand into the available space.

```
hbox <- gtkHBox()
hbox$packStart(slider, expand=TRUE, fill = TRUE, padding = 5)
hbox$packStart(spinbutton, expand = FALSE, padding = 5)
```

8.4 Progress reporting

Progress bars

It is common to use a progress bar to indicate the progress of a long-running computation. This is implemented by `GtkProgressBar`. A text label describes the current operation, and the progress bar communicates the fraction completed:

```
window <- gtkWindow(); window$setTitle("Progress bar example")
progress_bar <- gtkProgressBar()
window$add(progress_bar)
#
progress_bar$setText("Please be patient...")
for(i in 1:100) {
  progress_bar$setFraction(i/100)
  Sys.sleep(0.05) ## replace with a step in the process
}
progress_bar$setText("All done.")
```

We can indicate indefinite activity by periodically pulsing the bar:

```
progress_bar$pulse()
```

Spinners

Related to a progress bar is the `GtkSpinner` widget, which is a graphical heartbeat to assure the user that the application is still alive during long-running operations. Spinners are commonly found in web browsers. The basic usage is straightforward:

```
spinner <- gtkSpinner()
spinner$start()
spinner$stop()
```

8.5 Wizards

The `GtkAssistant` class provides a wizard widget for GTK+. The simplest setup is that one adds pages to the assistant object that are navigated in a linear manner. In our example, we override this.

Wizard pages have a certain type, which must be declared. These are enumerated in `GtkAssistantPageType` and set by `setPageType`. The last page must be of type "confirm", "summary", or "progress". Each wizard page has a content area and buttons. As well, each page in the assistant object has an optional side image, header image, and/or page title that can be customized. The buttons allow the user to navigate through the wizard. The content area of a wizard page is simply an instance of class `GtkWidget` (e.g., some container) and is added to the assistant through the `appendPage`, `insertPage`, or `prependPage` methods. Pages are referred to by the `GtkWidget` object or by their page index, 0-based. The forward button on a page must be made sensitive by calling `setPageComplete` with the widget and logical value.

Signals The cancel button emits a `cancel` signal that can be connected to for destroying the wizard widget. The `apply` signal is emitted on a page change. The `prepare` signal is emitted just before a page is made visible, which is needed to create the dynamically generated pages in our example.

Example 8.10: An `install.packages` wizard

This example wraps the `install.packages` function into a wizard with different pages for the (optional) selection of a CRAN mirror, the selection of the package to install, the configuration options provided, and feedback. In general, wizards are quite common for software installation.

First, we define our assistant and connect to its `cancel` signal:



Figure 8.6: An installation wizard programmed using GtkAssistant. This is page four, which allows options for a call to `install.packages` to be configured.

```
assistant <- gtkAssistant(show=FALSE)
assistant$setSizeRequest(500, 500)
gSignalConnect(assistant, "cancel",
               function(assistant) assistant$destroy())
```

Our pages will be computed dynamically. Here we populate the pages using box containers and specify their respective types:

```
pages <- lapply(1:5, gtkVBox, spacing=5, homogeneous = FALSE)
page_types <- c("intro", rep("confirm", 3), "summary")
sapply(pages, gtkAssistantAppendPage, object = assistant)
sapply(pages, gtkAssistantSetPageType, object = assistant,
       type=page_types)
```

We customize each page with a side logo.

```
image <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
sapply(pages, gtkAssistantSetPageSideImage, object=assistant,
       pixbuf = image)
```

When a page is about to be called, the `prepare` signal is emitted. In our handler, we check and see if it has any children. If not, we call a function to create the page lazily. These functions are stored in a list, so that we can refer to them by index.

```

populate_page <- list()
gSignalConnect(assistant, "prepare",
  function(assistant, page, data) {
    page_no <- which(sapply(pages, identical, page))
    if(!length(page$getChildren()))
      populate_page[[page_no]]()
  })

```

Although we do not show how to create the CRAN selection page (cf. Example 9.5 for a similar construction), we call `setForwardPageFunc` to set a function that will skip this page if it is not needed, i.e., if the mirror has already been selected. The callback simply returns an integer with the next page number based on the previous one.

```

assistant$setForwardPageFunc(function(page_index, data) {
  if(page_index == 0 && have_CRAN())
    2L
  else
    as.integer(page_index + 1)
}, data=NULL)

```

We have a few script globals that allow us to pass data between pages:

```

CRAN_package <- NA
install_options <- list() #type, dependencies, lib

```

We now show how some of the pages are populated. The initial screen is simply a label with a welcome message.

```

populate_page[[1]] <- function() {
  assistant$setPageTitle(pages[[1]], "Install a CRAN package")
  pages[[1]]$packStart(label <- gtkLabel())
  pages[[1]]$packStart(gtkLabel(), expand=TRUE) # a spring

  label$setMarkup(paste(
    "<span font='x-large'>Install a CRAN package</span>",
    "This wizard will help install a package from",
    "<b>CRAN</b>. If you have not already specified a",
    "CRAN repository, you will be prompted to do so.",
    sep="\n"))
  assistant$setPageComplete(pages[[1]], TRUE)
}

```

We skip showing the pages to select a CRAN site and a package, as they are based on the forthcoming `GtkTreeView` class. On the fourth page (cf. Figure 8.6 for a realization) is a summary of the package taken from CRAN and a chance for the user to configure a few options for the `install.packages` function.

```

populate_page[[4]] <- function() {

```

```
assistant$setPageTitle(pages[[4]], "Install a CRAN package")
##
get_desc <- function(pkgname) {
  o <- "http://cran.r-project.org/web/packages/%s/%s"
  x <- readLines(sprintf(o, pkgname, "DESCRIPTION"))
  f <- tempfile(); cat(paste(x, collapse="\n"), file=f)
  read.dcf(f)
}
desc <- get_desc(CRAN_package)
#
label <- gtkLabel()
label$setLineWrap(TRUE)
label$setWidthChars(40)
label$setMarkup(paste(
  sprintf("Install package: <b>%s</b>", desc[1,'Package']),
  "\n",
  sprintf("%s", gsub("\\n", " ", desc[1,'Description']),
  sep="\n"))
pages[[4]]$packStart(label)
##
table <- gtkTable()
pages[[4]]$packStart(table, expand=FALSE)
pages[[4]]$packStart(gtkLabel(), expand=TRUE)

##
combo <- gtkComboBoxNewText()
pkg_types <- c("source", "mac.binary", "mac.binary.leopard",
              "win.binary", "win64.binary")
sapply(pkg_types, combo$appendText)
combo$setActive(which(getOption("pkgType") == pkg_types)-1)
gSignalConnect(combo, "changed", function(combo, ...) {
  cur <- 1L + combo$getActive()
  install_options[['type']] <-< pkg_types[cur]
})
table$attachDefaults(gtkLabel("Package type:"), 0, 1, 0, 1)
table$attachDefaults(combo, 1, 2, 0, 1)

##
checkBox <- gtkCheckBox()
checkBox$setActive(TRUE)
gSignalConnect(checkButton, "toggled", function(ck_btn) {
  install_options$dependencies <-< ck_btn$getActive()
})
table$attachDefaults(gtkLabel("Install dependencies"),
                    0, 1, 1, 2)
table$attachDefaults(checkButton, 1, 2, 1, 2)
```

```

##
file_chooser <- gtkFileChooserButton("Select directory...",
                                     "select-folder")
file_chooser$setFilename(.libPaths()[1])
gSignalConnect(file_chooser, "selection-changed",
               function(file_chooser) {
                 dir <- file_chooser$getFilename()
                 install_options[['lib']] <-< dir
               })
table$attachDefaults(gtkLabel("Where"), 0, 1, 2, 3)
table$attachDefaults(file_chooser, 1, 2, 2, 3)
## align labels to right and set spacing
sapply(table$getChildren(), function(child) {
  widget <- child$getWidget()
  if(is(widget, "GtkLabel")) widget['xalign'] <- 1
})
table$setColSpacing(0L, 5L)
##
assistant$setPageComplete(pages[[4]], TRUE)
}

```

Our last page, where the selected package is installed, would naturally be of type `progress`, but there is no means to interrupt the flow of `install.packages` to update the page. A better application would need to reimplement the `install.packages` functionality. Instead we just set a message once the package install attempt is finished.

```

populate_page[[5]] <- function() {
  assistant$setPageTitle(pages[[5]], "Done")
  install_options$pkgs <- CRAN_package
  out <- try(do.call("install.packages", install_options),
            silent=TRUE)

  label <- gtkLabel(); pages[[5]]$packStart(label)
  if(!inherits(out, "try-error")) {
    label$setMarkup(sprintf("Package %s was installed.",
                           CRAN_package))
  } else {
    label$setMarkup(paste(sprintf("Package %s, failed install",
                                 CRAN_package),
                          paste(out, collapse="\n"),
                          sep="\n"))
  }

  assistant$setPageComplete(pages[[5]], FALSE)
}

```

To conclude, we populate the first page and call the assistant's show method:

```
populate_page[[1]]()
assistant$show()
```

8.6 Embedding R graphics

The package `cairoDevice` is an R graphics device based on the Cairo graphics library. It supports alpha-blending and antialiasing and reports user events through the `getGraphicsEvent` function. 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.

This is the simplest usage:

```
library(cairoDevice)
device <- gtkDrawingArea()
asCairoDevice(device)
##
window <- gtkWindow(show=FALSE)
window$add(device)
window$showAll()
plot(mpg ~ hp, data = mtcars)
```

In the above, we create the `GtkDrawingArea`, coerce it to a Cairo-based graphics device, and then place it in a window. Example 7.4 goes further by embedding the drawing area into a scrolled window to support zooming and panning.

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 could then be composited with other images when displayed. Example 8.5 generates an icon by pointing the device to a pixmap. Finally, passing a `GtkPrintContext` to `asCairoDevice` allows printing R graphics through the GTK+ printing dialogs.

Example 8.11: Printing R graphics

This example will show how to use the printing support in GTK+ for printing an R plot.

A print operation is encapsulated by `GtkPrintOperation`:

```
print_op <- gtkPrintOperation()
```

A print operation may perform several different actions: print directly, print through a dialog, show a print preview and export to a file. Before performing any such action, we need to implement the rendering of our document into printed form. This is accomplished by connecting to the `draw-page` signal. The handler is passed a `GtkPrintContext`, which contains the target Cairo context. In general, one would call Cairo functions to render the document, which is beyond our scope. In this case, though, we can pass the context directly to `cairoDevice` for rendering the R plot:

```
gSignalConnect(print_op, "draw-page",
               function(print_op, context, page_nr) {
                 asCairoDevice(context)
                 plot(mpg ~ wt, data = mtcars)
               })
```

The final step is to run the operation to perform one of the available actions. In this example, we launch a print dialog:

```
print_op$run(action = "print-dialog", parent = NULL)
```

When the user confirms the dialog, the `draw-page` handler is invoked, and the rendered page is sent to the printer.

Example 8.12: The `manipulate` package in `RGtk2`

`RStudio`TM is an excellent IDE for R that provides a similar interface whether run on any of its supported operating systems or through a web browser. Accompanying the IDE is an R package `manipulate` that provides a convenient means to create simple graphical interfaces for plotting. As `RStudio` leverages web technologies to render its widgets and there is no public interface, the package is not available for non-`RStudio` users. Too bad. This example shows how we can use `RGtk2` to provide a similar interface. In the example, we borrow liberally from the `manipulate` code, which is released under an AGPL license. Although we don't show the entire code here, the `ProgGUIinR` package contains it all.

The `manipulate` package uses environments to store state etc. Here we use reference classes, as they allow for a more structured programming interface.

A typical use of `manipulate` (Figure 8.7) is along the lines of the following example from the `manipulate` help pages:

```
manipulate(## expression
           plot(cars, xlim = c(x.min, x.max), type = type,
                axes = axes, ann = label),
           ## controls
           x.min = slider(0, 15),
           x.max = slider(15, 30, initial = 25),
           type = picker("p", "l", "b", "c", "o", "h", "s"),
           axes = checkbox(TRUE, label = "Draw Axes"),
```

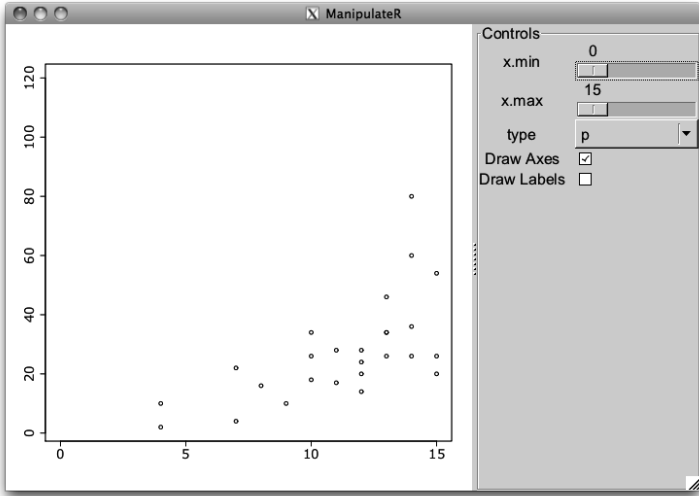


Figure 8.7: An implementation of RStudio'sTM `manipulate` package in RGtk2.

```
label = checkbox(FALSE, label = "Draw Labels")
)
```

The first argument is an expression, possibly containing parameters, that produces a plot. The other arguments create widgets that control the parameter values in the plotting expression. There are three basic controls: a slider, a picker (combo box), and a checkbox. The constructors have a terse but simple set of arguments. A main task ahead will be mapping these controls to one of GTK+'s widgets.

For now, we begin by defining our `Manipulate` class to have two properties: one to hold the expression and the other to hold a list of controls.

```
Manipulate <- setRefClass("Manipulate",
  fields=list(
    .code="ANY",
    .controls="list"
  ))
```

When one of the controls is changed, the entire plot will be redrawn. The following handler will be assigned to each control. Note that each control is expected to provide its own `get_value` method.

```
Manipulate$methods(
  get_values = function() {
    sapply(.controls,
      function(control) control$get_value(),
      simplify=FALSE) # return a list
```

```

    },
    change_handler = function(...) {
      "Evaluate code with current values"
      values <- get_values()
      result <- withVisible(eval(.code, envir=values))
      if (result$visible) {
        eval(print(result$value))
      }
    }
  })

```

The `execute` method is called after initialization to set up the GUI. We use a `GtkHPaned` instance to allow the user to adjust the space between the graphic device and the controls frame. Each control is expected to provide a `make_gui` method.

```

Manipulate$methods(
  execute=function() {
    "Make the GUI"
    window <- gtkWindow(show=FALSE)
    window$setTitle("ManipulateR")
    ## Set up graphic device
    hpaned <- gtkHPaned()
    window$add(hpaned)
    device <- gtkDrawingArea()
    device$setSizeRequest(480, 480)
    asCairoDevice(device)
    hpaned$add(device)
    ## Controls frame
    frame <- gtkFrame("Controls")
    control_table <- gtkTableNew()
    control_table$setHomogeneous(FALSE)
    control_table['column-spacing'] <- 10
    ## insert horizontal strut
    control_table$attach(strut <- gtkHBox(), 1,2,0,1,
                        xoptions="", yoptions="shrink")
    strut$setSizeRequest(75, -1)
    frame$add(control_table)
    hpaned$add(frame)
    ## add each control
    sapply(.controls, function(control) {
      control$make_gui(cont=control_table,
                      handler=.self$change_handler)
    })
    window$show()
    change_handler() # initial
  })

```


The `control_table` is used hold the respective controls. We added a strut to request a minimum width for the second column, as otherwise the slider controls can render too narrowly.

The `initialize` method calls a function provided by the `manipulate` package to pick the controls out of the `...` argument. The `validate_controls` method is not shown but simply borrows code from the package to do some error checking, ensuring the controls are defined properly.

```
Manipulate$methods(  
  initialize = function(code, ...) {  
    controls <- resolveVariableArguments(list(...))  
    initFields(.code = code,  
              .controls = controls)  
    validate_controls()  
    callSuper()  
  })
```

We now provide a constructor allowing access to our class.

```
manipulate <- function('_expr', ...) {  
  manip <- Manipulate$new(substitute('_expr'), ...)  
  manip$execute()  
}
```

There are three main controls, but perhaps more could be added. We give ourselves the flexibility to expand by creating a base class for a control that can be subclassed. We define the class below. The properties are `l`, to store a list of arguments (a legacy of the original code); `widget`, to store the widget; `label`, to hold the label for the control; and `initial`.

```
ManipulateControls <- setRefClass("ManipulateControls",  
  fields=list(  
    l="list",  
    widget = "ANY",  
    label="ANY",  
    initial="ANY"  
  ))
```

The main interface for a control requires three methods: `validate_inputs` (to ensure the control is defined properly, the previously noted `get_value`, and `make_gui` (defined separately).

```
ManipulateControls$methods(  
  validate_inputs = function(...) {  
    "Validate input code"  
  },  
  get_value = function(...) {  
    "Get value of widget"  
  })
```

The `make_gui` method has two tasks: to define the widget instance and to add the widget to the GUI. This is done in the base class. The label and widget are added as a row to a `GtkTable` instance.

```
ManipulateControls$methods(make_gui = function(cont) {
  "Create widget, then add to table"
  ## cont a GtkTable instance
  nrows <- cont['n-rows']
  label_widget <- gtkLabel(label)
  label_widget['xalign'] <- 1
  cont$attach(label_widget, 0, 1, nrows, nrows + 1,
              xoptions = "shrink", yoptions="shrink"
              )
  cont$attach(widget, 1, 2, nrows, nrows + 1,
              xoptions = c("expand", "fill"),
              yoptions = "")
})
```

The slider constructor just creates an instance of a soon-to-be-defined subclass of the `ManipulateControls` class. The arguments follow RStudio's.

```
slider <- function(min, max, initial = min, label=NULL,
                  step = -1, ticks = TRUE) {
  Slider$new(min, max, initial = initial, label = label,
            step = step, ticks = ticks)
}
```

The `Slider` class has no new properties:

```
Slider <- setRefClass("Slider",
                    contains = "ManipulateControls")
```

The `initialize` method simply creates a list and sets some properties. This follows the setup of the original package.

```
Slider$methods(
  initialize = function(min, max, initial = min,
                        label = NULL, step = -1, ticks = TRUE, ...) {
    validate_inputs(min, max, initial, step, ticks)
    ## create slider and return it
    slider <- list(type = 0,
                  min = min,
                  max = max,
                  step = step,
                  ticks = ticks)
    initFields(l = slider, label = label,
              initial = initial)
    callSuper()
  })
```

Our `make_gui` method basically defines the widget, turning the arguments of the constructor into those for the GTK+ widget. It then calls the same method from the superclass to lay out the widget. Here we define a slider and initialize it using the values in the list, `l`. The handler is the change handler passed in from a `Manipulate` instance.

```
Slider$methods(  
  make_gui = function(cont, handler, ...) {  
    widget <- gtkHScale(min = l$min, max = l$max,  
                       step = l$step)  
    widget$setValue(initial)  
    gSignalConnect(widget, "value-changed", handler)  
    callSuper(cont)  
  },  
  get_value = function() {  
    as.numeric(widget$getValue())  
  })
```

The picker and checkbox functions (and their classes) are similarly defined. For example, for the `Checkbox` class, the three main methods are given by:

```
Checkbox$methods(  
  initialize = function(initial=FALSE, label= NULL) {  
    validate_inputs(initial, label)  
    checkbox <- list(type = 2)  
    initFields(l = checkbox, label = label,  
              initial = initial)  
    callSuper()  
  },  
  make_gui = function(cont, handler, ...) {  
    widget <- gtkCheckButton() # no label  
    widget$setActive(initial)  
    gSignalConnect(widget, "toggled", handler)  
    callSuper(cont)  
  },  
  get_value = function() widget['active']  
)
```

We don't provide a label to the check button, as one is provided in the table.

8.7 Drag-and-drop

A drag-and-drop operation is the movement of data from a source widget to a target widget. In GTK+ the source widget serializes the selected item as MIME data, and the destination interprets that data to perform some operation, often creating an item of its own. Our task is to configure the

source and destination widgets, so that they listen for the appropriate events and understand each other. As a trivial example, we allow the user to drag the text from one button to another.

Initiating a drag

When a drag-and-drop is initiated, different types of data may be transferred. We need to define a target type for each type of data, as a `GtkTargetEntry` structure:

```
TARGET.TYPE.TEXT <- 80          # our enumeration
TARGET.TYPE.PIXMAP <- 81
widgetTargetTypes <-
  list(text = gtkTargetEntry("text/plain", 0,
    TARGET.TYPE.TEXT),
    pixmap = gtkTargetEntry("image/x-pixmap", 0,
    TARGET.TYPE.PIXMAP))
```

The first component of `GtkTargetEntry` is the name, which is often a MIME type. The flags come next, which are usually left at 0, and finally we specify an arbitrary identifier for the target. We will use only the "text" target in this example.

We construct a button and call `gtkDragSourceSet` to instruct it to act as a drag source:

```
window <- gtkWindow(); window['title'] <- "Drag Source"
drag_source_widget <- gtkButton("Text to drag")
window$add(drag_source_widget)
gtkDragSourceSet(drag_source_widget,
  start.button.mask=c("button1-mask", "button3-mask"),
  targets=widgetTargetTypes[["text"]],
  actions="copy")
```

The `start.button.mask`, with values from `GdkModifierType`, indicates the modifier buttons that need to be pressed to initiate the drag. The allowed target is "text" in this case. The `actions` argument lists the supported actions, such as copy or move, from the `GdkDragAction` enumeration.

When a drag is initiated, we will receive the `drag-data-get` signal, which needs to place some data into the passed `GtkSelectionData` object:

```
gSignalConnect(drag_source_widget, "drag-data-get",
  function(widget, context, sel, tType, eTime) {
    sel$setText(widget$getLabel())
  })
```

If we had allowed the move action, we would also need to connect to `drag-data-delete`, in order to delete the data that was moved away.

Handling drops

In a separate window from the drag source button, we construct another button and call `gtkDragDestSet` to mark it as a drag target:

```
window <- gtkWindow(); window['title'] <- "Drop Target"
drop_target_widget <- gtkButton("Drop here")
window$add(drop_target_widget)
gtkDragDestSet(drop_target_widget,
               flags="all",
               targets=widgetTargetTypes[["text"]],
               actions="copy")
```

The signature is similar to that of `gtkDragSourceSet`, except for the `flags` argument, which indicates which operations, of the set motion, highlight, and drop, GTK+ will handle with reasonable default behavior. Specifying `all` is the most convenient course, in which case we need only to implement the extraction of the data from the `GtkSelectionData` object. For a drop to occur, there must be a non-empty intersection between the targets passed to `gtkDragSourceSet` and those passed to `gtkDragDestSet`.

When data is dropped, the destination widget emits the `drag-data-received` signal. The handler is responsible for extracting the dragged data from selection and performing some operation with it. In this case, we set the text on the button:

```
gSignalConnect(drop_target_widget, "drag-data-received",
               function(widget, context, x, y, sel, tType, eTime) {
                 dropdata <- sel$getText()
                 widget$setLabel(rawToChar(dropdata))
               })
```

The `context` argument is a `GdkDragContext`, containing information about the drag event. The `x` and `y` arguments are integer valued and represent the position in the widget where the drop occurred. The text data is returned by `getText` as a raw vector, so it is converted with `rawToChar`.

RGtk2: Widgets Using Data Models

Many widgets in GTK+ use the model-view-controller (MVC) paradigm. For most, like the button widget, 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.

9.1 Displaying 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` can be implemented in any number of ways. GTK+ provides simple in-memory implementations for hierarchical and nonhierarchical data. 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. The `RGtkDataFrame` class implements `GtkTreeModel` on top of an R data frame. Compared to the model implementations built into GTK+, `RGtkDataFrame` affords the R programmer the benefits of improved speed, convenience and familiarity.

For nonhierarchical data, this is usually the model of choice, so we discuss it first. Populating an `RGtkDataFrame` is far faster than for a GTK+ model, because data is retrieved from the data frame on demand. There is no need to copy the data row by row into a separate data structure. Such an approach would be especially slow if implemented as a loop in

R.¹ The constructor `rGtkDataFrame` 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 cannot be removed by assigning values to NULL, and column types should not be changed. These limitations are inherent in the design of GTK+: columns cannot be removed from GtkTreeModel, and views expect the data type to remain the same.`

Example 9.1: Defining and manipulating an `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
```

As with a data frame, 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, we 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])
```

Replacing the data frame is the only way to remove rows, as this is not possible with the 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 treeview 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,

¹As is proved with `tc1tk`, where this is needed.

it is a table. If the rows are nested, it is a tree table, where every node has values on the same columns.

A treeview 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 treeview 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:

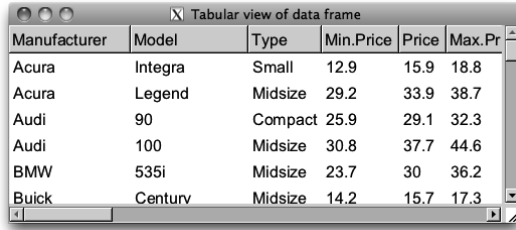
```
column <- gtkTreeViewColumn()
column$title("Manufacturer")
cell_renderer <- gtkCellRendererText()
column$packStart(cell_renderer)
column$addAttribute(cell_renderer, "text", 0)
view$insertColumn(column, 0)
```

A column with the title “Manufacturer” is inserted at the first position (0-based). For displaying a simple data frame, we need only 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 the `GtkCellRendererText` class. There are several properties that control how the text is rendered. An *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 = 2 - 1) # second column
```

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.



Manufacturer	Model	Type	Min.Price	Price	Max.Pr
Acura	Integra	Small	12.9	15.9	18.8
Acura	Legend	Midsize	29.2	33.9	38.7
Audi	90	Compact	25.9	29.1	32.3
Audi	100	Midsize	30.8	37.7	44.6
BMW	535i	Midsize	23.7	30	36.2
Buick	Centurv	Midsize	14.2	15.7	17.3

Figure 9.1: A GtkTreeView instance shown with a scrolled window.

Displaying the entire Cars93 data frame is not much different. 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,
       position = -1,
       title = colnames(model),
       cell = list(gtkCellRendererText()),
       text = seq_len(ncol(model)) - 1
       )
```

Figure 9.1 shows the view within a scrollable window:

```
window <- gtkWindow()
window$title <- "Tabular view of data frame"
scrolled_window <- gtkScrolledWindow()
window$add(scrolled_window)
scrolled_window$add(view)
```

Manipulating view columns The GtkTreeView widget is essentially a collection of columns. Columns are added to the treeview 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 treeview 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 Treeviews have several special features, including sorting, incremental search, and drag-and-drop reordering. Sorting is discussed in Section 9.1. 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 treeview will pop up 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 interact directly 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 table, 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 can be expressed as either a numeric vector or a string, using `gtkTreePathNewFromIndices` or `gtkTreePathNewFromString`, respectively. For a flat table model, there is only one integer in the sequence:

```
second_row <- gtkTreePathNewFromIndices(1)
```

Referring to a row in a hierarchy is slightly more complex:

```
abc_path <- gtkTreePathNewFromIndices(c(0, 2, 1))
abc_path <- gtkTreePathNewFromString("0:2:1")
```

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.

Iterators The second means of row indexing is through an iterator, `GtkTreeIter`, which is better suited for traversing a model. 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, in the sense that they are invalidated when their source is modified. An iterator is often updated by reference, behavior that is atypical in R programming.

While a tree path is an intuitive, transparent row index, an iterator 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 visit each row in sequence:

```
iter <- model$getIterFirst()
manufacturer <- character()
while(iter$retval) {
  manufacturer <- c(manufacturer, model$get(iter$iter,0)[[1]])
  iter$retval <- model$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. This iterator design also prevents the use of the apply functions (R's iterators), 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$iterNChildren(NULL)
manufacturer <- sapply(seq(nrows) - 1L, function(i) {
  iter <- model$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 neither intuitive nor fast, so the benefits of `RGtkDataFrame` should be obvious.

Converting between paths and iterators We can convert between paths and iterators. 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 final point: `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, `GtkTreeRowReference` is used.

Selection

There are multiple modes of user interaction with a treeview: 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 9.1. Here, we limit our discussion to selection of rows.

GTK+ provides the class `GtkTreeSelection` to manage row selection. Every treeview 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. For example, we create a view and limit it to single selection:

```
model <- rGtkDataFrame(mtcars)
view <- gtkTreeView(model)
selection <- view$getSelection()
selection$setMode("single")
```

When only a single selection is possible, the method `getSelected` returns the selected row as a 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. If our treeview is shown and a selection made, this code will return the value in the first column:

```
selected <- selection$getSelected()
with(selected, model$getValue(iter, 0)$value)
```

```
[1] 21.4
```

When multiple selection is permitted, then the method `getSelectedRows` returns a list with the `model` and `retval`, a list of tree paths.

To respond to a selection, connect to the changed signal on `GtkTreeSelection`. Upon a selection, this handler will print the selected values in the first column:

```
gSignalConnect(selection, "changed", function(selection) {
  selected_rows <- selection$getSelectedRows()
  if(length(selected_rows$retval)) {
    rows <- sapply(selected_rows$retval,
                   gtkTreePathGetIndices) + 1L
    selected_rows$model[rows, 1]
  }
})
```

When a row is not editable, then the double-click event or a keyboard command triggers the row-activated signal for the treeview. 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 treeview 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 treeview through a command like

```
sapply(view$getColumns(), 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 necessary only to inform each view column of the model column to use for sorting when the header is clicked:

```
column <- view$getColumn(0)
column$setSortColumnId(0)
```

In the above, clicking on the header of the first view column 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, "ascending")
```

Some models, however, such as the `GtkTreeModelFilter` introduced in the next section, do not implement `GtkTreeSortable`. 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` for this:

```
model <- rGtkDataFrame(Cars93)
sorted_model <- gtkTreeModelSortNewWithModel(model)
```



Manufacturer	Model	Type	Min.Price	Price	Max.Price	MPG.city
Volkswagen	Passat	Compact	17.6	20	22.4	21
Pontiac	Sunbird	Compact	9.4	11.1	12.8	23
Dodge	Spirit	Compact	11.9	13.3	14.7	22
Ford	Festiva	Small	6.9	7.4	7.9	31
Ford	Escort	Small	8.4	10.1	11.9	23
Geo	Metro	Small	6.7	8.4	10	46
Honda	Civic	Small	8.4	12.1	15.8	42
Mazda	Protege	Small	10.9	11.6	12.3	28

Figure 9.2: When a sortable model is passed to the treeview, we can click on the column headers to sort the data. The "Type" column has a custom sort function applied.

```
view <- gtkTreeView(sorted_model)
mapply(view$insertColumnWithAttributes,
       position = -1,
       title = colnames(model),
       cell = list(gtkCellRendererText()),
       text = seq_len(ncol(model)) - 1)
sapply(seq_len(ncol(model)), function(i)
       view$getColumn(i - 1)$setSortColumnId(i - 1))
```

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 a special sort for the Type variable can be implemented (Figure 9.2).

```
f <- function(model, iter1, iter2, user.data) {
  types <- c("Compact", "Small", "Sporty", "Midsize",
            "Large", "Van")
  column <- user.data
  val1 <- model$getValue(iter1, column)$value
  val2 <- model$getValue(iter2, column)$value
  as.integer(match(val1, types) - match(val2, types))
}
sorted_model$setSortFunc(sort.column.id = 3 - 1, sort.func=f,
                        user.data = 3 - 1)
```

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 whether a row should be visible. The index of that column

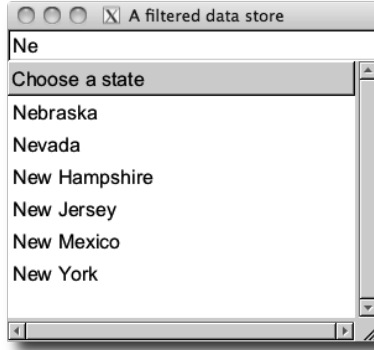


Figure 9.3: Example of a data model filtered by values typed into a text-entry widget.

is passed to the filter model, which provides only those rows where the filter column is TRUE.

This is the basic usage:

```
DF <- Cars93
model <- rGtkDataFrame(cbind(DF, .vis=rep(TRUE, nrow(DF))))
filtered_model <- model$filter()
filtered_model$setVisibleColumn(length(DF)) # 0-based
view <- gtkTreeView(filtered_model)
## Adjust filter
model[, ".vis"] <- DF$MPG.highway >= 30
```

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. To customize filtering, we can register a function with `setVisibleFunc`. The callback, given a row pointer, should return TRUE if the row passes the filter (see Example 9.4). A filter model can be treated as any other tree model, including attachment to a `GtkTreeView`.

Example 9.2: 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))
```

```
model <- rGtkDataFrame(DF)
```

The filtered model needs to have the column specified that contains the logical values; in this example, it is the last column.

```
filtered_model <- model$filter()
filtered_model$setVisibleColumn(ncol(DF) - 1)      # offset
view <- gtkTreeView(filtered_model)
```

Next, we create a basic view of a single column:

```
view$insertColumnWithAttributes(0, "Col",
                               gtkCellRendererText(), text = 0)
```

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.

```
entry <- gtkEntry()
gSignalConnect(entry, "changed", function(entry, user.data) {
  pattern <- entry$getText()
  DF <- user.data$getModel()
  values <- DF[, "state.name"]
  DF[, "visible"] <- grepl(pattern, values)
}, data=filtered_model)
```

Figure 9.3 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 9.3), 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 `cell-background` property) might be common to all rows in the column and thus is set explicitly, without use of an attribute:

```
cell_renderer <- gtkCellRendererText()
cell_renderer['cell-background'] <- "gray"
```

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 `GtkCellRendererText` displays 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. For example, we display right-aligned text in a Helvetica font:

```
cell_renderer <- gtkCellRendererText()
cell_renderer['xalign'] <- 1 # default 0.5 = centered
cell_renderer['family'] <- "Helvetica"
```

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 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. These arguments do not include the `treeview` object nor the column index, so these should be provided by some other means, e.g., from the enclosing environment of the handler. For example, here is how we can update an `RGtkDataFrame` model from within the callback:

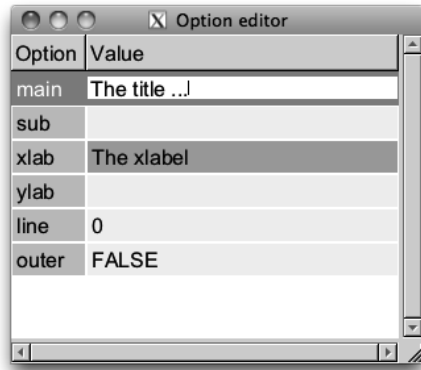


Figure 9.4: A treeview used to gather arguments for a call to `title`.

```
cell_renderer['editable'] <- TRUE
gSignalConnect(cell_renderer, "edited",
  f=function(cell_renderer, path, newtext, user.data) {
    i <- as.numeric(path) + 1
    j <- user.data$column
    model <- user.data$model
    model[i, j] <- newtext
  }, data=list(model=store, column=1))
```

Before using editable cells to create a data frame editor, we should see if the editor provided by the `gtkDfEdit` in the `RGtk2Extras` package satisfies the requirements.

Users may expect that once a cell is edited, the next cell is then set up to be edited. In order to do this, we must advance the cursor and activate editing of the next cell. For `GtkTreeView`, this is implemented by the `setCursor` method.

Example 9.3: Using a table to gather arguments

This example shows one way to gather arguments or options using an editable cell in a table, rather than a separate text-entry widget. Tables can provide compact entry areas in a familiar interface.

For this example we collect values for arguments to the `title` function. We first create a data frame with the argument name and default value, along with some additional values:

```
opts <- c("main", "sub", "xlab", "ylab", "line", "outer")
DF <- data.frame(option = opts,
  value = c("", "", "", "", "0", "FALSE"),
  class = c(rep("character", 4), "integer", "logical"),
  edit_color = rep("gray95", 6),
```

```
dirty = rep(FALSE, 6),
stringsAsFactors = FALSE)
```

Unfortunately, we need to coerce the default values to character, in order to store them in a single column. We preserve the class in the class column, for coercion later. The edit_color and dirty columns are related to editing and explained later.

Now we create our model and configure the first column:

```
model <- rGtkDataFrame(DF)
view <- gtkTreeView(model)
##
cell_renderer <- gtkCellRendererText()
cell_renderer['background'] <- 'gray80'
view$insertColumnWithAttributes(position = -1,
                               title = "Option",
                               cell = cell_renderer,
                               text = 1 - 1)
```

The first column has a special background color, specified below, which indicates that the cells are not editable. The second column is editable and has a background color that is state dependent and indicates whether a cell has been edited (The xlab column in Figure 9.4):

```
cell_renderer <- gtkCellRendererText()
cell_renderer['editable'] <- TRUE
view$insertColumnWithAttributes(position = -1,
                               title = "Value",
                               cell = cell_renderer,
                               text = 2 - 1,
                               background = 4 - 1
                               )
```

To attach the view to the model, we connect the cell renderer to the edited signal. Here we use the class value to format the text and then set the background color and dirty flag of the entry. The latter allows us to easily find the values which were edited.

```
gSignalConnect(cell_renderer, "edited",
              function(cell_renderer, path, new.text, user.data) {
                model <- user.data$model
                i <- as.numeric(path) + 1; j <- user.data$column
                val <- as(new.text, model[i, 'class'])
                model[i,j] <- as(val, "character")
                model[i, 'dirty'] <- TRUE # mark dirty
                model[i, 'edit_color'] <- 'gray70' # change color
              }, data=list(model=model, column=2))
```

A simple window displays our GUI.

```

window <- gtkWindow(show=FALSE)
window['title'] <- "Option editor"
window$setSizeRequest(300,500)
scrolled_window <- gtkScrolledWindow()
window$add(scrolled_window)
scrolled_window$add(view)
window$show()

```

Implementing this dialog into a GUI requires writing a function to map the model values into the appropriate call to the `title` function. The `dirty` flag makes this easy, but this is a task we do not pursue here. Instead we add a bit of extra detail by providing a tooltip.

Tooltips For this example, our function has built-in documentation. Below, we use an internal function from the `helpr` package² to extract the description for each of the arguments. We leave this in a list, `descs`, for later lookup.

```

require(helpr, quietly=TRUE)
package <- "graphics"; topic <- "title"
rd <- helpr:::parse_help(helpr:::pkg_topic(package, topic),
                        package = package)
descs <- rd$params$args
names(descs) <- sapply(descs, function(i) i$param)

```

For many widgets, adding a tooltip is as easy as calling `setTooltipText`. However, it is more complicated in a treeview, as each cell should get a different tip. To add tooltips to the treeview we first indicate that we want tooltips, then connect to the `query-tooltip` signal:

```

view["has-tooltip"] <- TRUE
gSignalConnect(view, "query-tooltip",
  function(view, x, y, key_mode, tooltip, user.data) {
    out <- view$getTooltipContext(x, y, key_mode)
    if(out$retval) {
      model <- view$getModel()
      i <- as.numeric(out$path$toString()) + 1
      val <- model[i, "option"]
      txt <- descs[[val]]$desc
      txt <- gsub("code>", "b>", txt) # no code in Pango
      tooltip$setMarkup(txt)
    }
    out$retval
  })

```

²It is important to note that we are calling internal routines of a package still under active development, which in turn relies on volatile features of R. In general, such practice can lead to maintenance headaches. The purpose of this example is only to provide a natural demonstration of tooltips on a treeview.

Within this callback we check to see if we have the appropriate context (we are in a row), then, if so, use the path to find the description to set in the tooltip. The descriptions use HTML for markup, but the tooltip uses only Pango. As the code tag is not PANGO, we change to a bold tag using `gsub`.

Combo and spin cell 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.

```
cell_renderer <- gtkCellRendererCombo()
model <- rGtkDataFrame(state.name)
cell_renderer['model'] <- model
cell_renderer['text-column'] <- 0
cell_renderer['editable'] <- TRUE # needed
```

The spin button editor is configured by setting a `GtkAdjustment` on the `adjustment` property.

The changed signal is emitted when an item is selected in the combo box. The spin cell renderer inherits the edited signal from `GtkCellRendererText`.

Pixbuf cell renderers To display an image in a cell, `GtkCellRendererPixbuf` is appropriate. The image is specified through one of these properties: `stock-id`, a stock identifier; `icon-name`, the name of a themed icon; or `pixbuf`, an actual `GdkPixbuf` object, holding an image in memory. Using a list, we can store a `GdkPixbuf` in a `data.frame`, and thus an `RGtkDataFrame`. This is demonstrated in the next example.

Example 9.4: A variable selection widget

This example shows how to create a GUI for selecting variables from a data frame. The GUI is based on two lists. The left one indicates the variables that can be selected, and the right shows the variables that have been selected. An icon, indicating the variable type, is placed next to the variable name (Figure 9.5). A similar mechanism is part of the SPSS model specification GUI of Figure 1.4. For illustration purposes we use the `Cars93` data set.

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

First, we render an icon for each variable. The `make_icon` function from the `ProgGUIinR` package creates an icon as a grid object, which we render with `cairoDevice`:

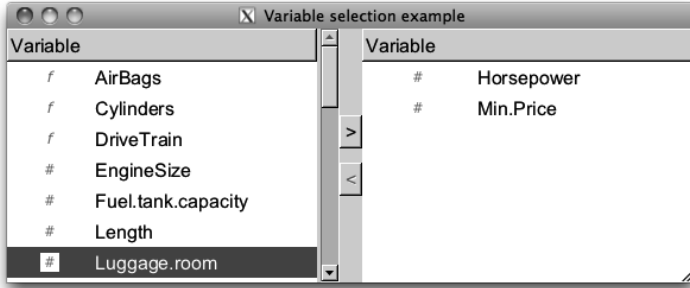


Figure 9.5: Illustration of an interface to select one or more variables. An icon is used in the table view to indicate the variable type.

```
make_icon_pixmap <- function(x, ...) {
  require(grid); require(cairoDevice)
  pixmap <- gdkPixmap(drawable = NULL, width = 16, height=16,
                      depth = 24)
  asCairoDevice(pixmap)
  grid.newpage()
  grid.draw(make_icon(x))
  dev.off()
  gdkPixbufGetFromDrawable(NULL, pixmap, NULL, 0,0,0,0,-1,-1)
}
```

The two list views are based on the same underlying data model, which contains three columns: the variable name, the icon, and whether the variable has been selected. We construct the corresponding data frame and wrap it in a `RGtkDataFrame` instance:

```
model_df <- data.frame(Variables = I(sort(names(DF))),
                      icon = I(sapply(DF, make_icon_pixmap)),
                      selected = rep(FALSE, ncol(DF)))
model <- rGtkDataFrame(model_df)
```

The first view shows only unselected variables, while the other is limited to selected variables. Thus, each view will be based on a different filter:

```
selected_filter <- model$filter()
selected_filter$setVisibleColumn(2)
unselected_filter <- model$filter()
unselected_filter$setVisibleFunc(function(model, iter) {
  !model$get(iter, 2)[[1]]
})
```

The selected filter is relatively easy to define, using `selected` as the visible column. For the unselected filter, we need to define a custom visible function that inverts the selected column.

Next, we create a view for each filter:

```
views <- list()
views$unselected_view <- gtkTreeView(unselected_filter)
views$selected_view <- gtkTreeView(selected_filter)
##
sapply(views, function(view) {
  selection <- view$getSelection()
  selection$setMode('multiple')
})
```

Each cell needs to display both an icon and a label. This is achieved by packing two cell renderers into the column:

```
make_view_column <- function() {
  column <- gtkTreeViewColumn()
  column$setTitle("Variable")
  column$packStart(cell_renderer <- gtkCellRendererPixbuf())
  column$addAttribute(cell_renderer, "pixbuf", 1L)
  column$packStart(cell_renderer <- gtkCellRendererText())
  column$addAttribute(cell_renderer, "text", 0L)
  column
}
sapply(views, function(view)
  view$insertColumn(make_view_column(), 0))
```

For later use we extend the API for a treeview – one method to find the selected indices (1-based) and one to indicate if there is a selection:

```
## add to the gtkTreeView API for convenience
gtkTreeViewSelectedIndices <- function(object) {
  model <- object$getModel() # Filtered!
  paths <- object$getSelection()$getSelectedRows()$retval
  path_strings <- sapply(paths, function(i) {
    model$convertpathToChildPath(i)$toString()
  })
  if(length(path_strings) == 0)
    integer(0)
  else
    as.numeric(path_strings) + 1 # 1-based
}
## does object have selection?
gtkTreeViewHasSelection <-
  function(obj) length(obj$selectedIndices()) > 0
```

Now we create the buttons and connect to the `clicked` signal. The handler moves the selected values to the other list by toggling the selected variable:

```
buttons <- list()
buttons$unselect_button <- gtkButton("<")
buttons$select_button <- gtkButton(">")
toggleSelectionOnClick <- function(button, view) {
  gSignalConnect(button, "clicked", function(button) {
    message("clicked")
    ind <- view$selectedIndices()
    model[ind, "selected"] <- !model[ind, "selected"]
  })
}
sapply(1:2, function(i) toggleSelectionOnClick(buttons[[i]],
                                              views[[3-i]]))
```

We want our buttons to be sensitive only if there is a possible move. This is determined by the presence of a selection:

```
sapply(buttons, gtkWidgetSetSensitive, FALSE)
trackSelection <- function(button, view) {
  gSignalConnect(view$getSelection(), "changed",
                 function(x) button['sensitive'] <- view$hasSelection())
}
sapply(1:2, function(i) trackSelection(buttons[[i]],
                                      views[[3-i]]))
```

We now lay out our GUI using a horizontal box, into which we pack the views and a box holding the selection buttons. The views will be scrollable, so are placed in scrolled windows:

```
window <- gtkWindow(show=FALSE)
window$setTitle("Select variables example")
window$setDefaultSize(600, 400)
hbox <- gtkHBox()
window$add(hbox)
## scrollwindows
scrolls <- list()
scrolls$unselected_scroll <- gtkScrolledWindow()
scrolls$select_scroll <- gtkScrolledWindow()
mapply(gtkContainerAdd, object = scrolls, widget = views)
mapply(gtkScrolledWindowSetPolicy, scrolls,
       "automatic", "automatic")
## buttons
button_box <- gtkVBox()
centered_box <- gtkVBox()
button_box$packStart(centered_box, expand=TRUE, fill = FALSE)
centered_box$setSpacing(12)
```



```
sapply(buttons, centered_box$packStart, expand = FALSE)
##
hbox$packStart(scrolls$unselected_scroll, expand = TRUE)
hbox$packStart(button_box, expand = FALSE)
hbox$packStart(scrolls$selected_scroll, expand = TRUE)
```

Finally, we show the top-level window:

```
window$show()
```

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 then needs to connect to the toggled to update the model in response to changes in the active state.

```
cell_renderer <- gtkCellRendererToggle()
cell_renderer['activatable'] <- TRUE # cell can be activated
cell_renderer['active'] <- TRUE
gSignalConnect(cell_renderer, "toggled", function(w, path) {
  model$active[as.numeric(path) + 1] <- w['active']
})
```

To render the toggle as a radio button instead of a checkbox, set the `radio` property to `TRUE`. Again, the programmer is responsible for implementing the radio-button logic via the `toggled` signal.

Example 9.5: Displaying a checkbox column in a treeview

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 generalized 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.

```
old_packages <-
  old.packages()[,c("Package", "Installed", "ReposVer")]
DF <- as.data.frame(old_packages)
```

This function will be called on the selected rows. Here, we simply call `install.packages` to update the selected packages.

```
doUpdate <- function(old_packages)
  install.packages(old_packages$Package)
```

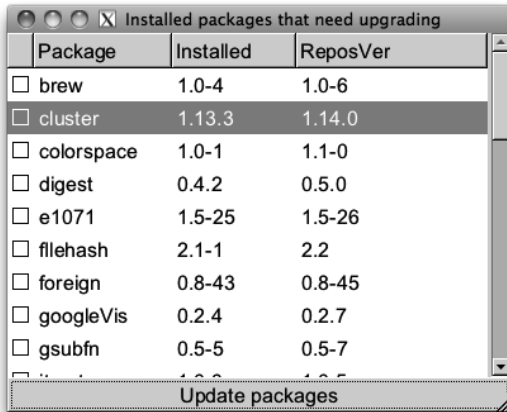


Figure 9.6: A GUI to select packages using checkboxes rendered with a `GtkCellRendererToggle` instance.

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`.

```
model <- rGtkDataFrame(cbind(DF, .toggle=rep(FALSE, nrow(DF))))
```

Our treeview shows each text column using a simple text-cell renderer, except for the first column that contains the checkboxes for selection.

```
view <- gtkTreeView()
cell_renderer <- gtkCellRendererToggle() # add toggle
view$insertColumnWithAttributes(0, "", cell_renderer,
                               active = ncol(DF))
cell_renderer['activatable'] <- TRUE
gSignalConnect(cell_renderer, "toggled",
               function(cell_renderer, path, user.data) {
                 view <- user.data
                 row <- as.numeric(path) + 1
                 model <- view$getModel()
                 n <- dim(model)[2]
                 model[row, n] <- !model[row, n]
               }, data=view)
```

The text columns are added in one go:

```
mapply(view$insertColumnWithAttributes, -1, colnames(DF),
       list(gtkCellRendererText(), text = seq_along(DF) - 1L))
```

Finally, we connect the store to the model.

```
view$setModel(model)
```

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.

```
button <- gtkButton("Update packages")
gSignalConnect(button, "clicked", function(button, data) {
  view <- data
  model <- view$getModel()
  old_packages <-
    model[model[, ncol(model)], -ncol(model), drop = FALSE]
  doUpdate(old_packages)
}, data=view)
```

Our basic GUI places the view into a box container that also holds the button to initiate the action.

```
window <- gtkWindow(show = FALSE)
window$setTitle("Installed packages that need upgrading")
window$setSizeRequest(300, 300)
vbox <- gtkVBox(); window$add(vbox)
scrolled_window <- gtkScrolledWindow()
vbox$packStart(scrolled_window, expand = TRUE, fill = TRUE)
scrolled_window$add(view)
scrolled_window$setPolicy("automatic", "automatic")
vbox$packStart(button, expand = FALSE)
window$show()
```

Progress cell renderers To communicate progress within a cell visually, 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. For example,

```
cell_renderer <- gtkCellRendererProgress()
cell_renderer["value"] <- 50
```

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 nontrivial task. Although the logic in the R print system often performs acceptably, there is certainly room for customization. One example is 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). Alternatively, we could preserve the integrity of the data and perform the conversion during rendering. This requires intercepting the model value before it is passed to the cell renderer.

In the specific case of `GtkTreeView`, it is possible to specify a callback that overrides this step. The callback, of type `GtkTreeCellDataFunc`, is passed arguments for the treeview column, the cell renderer, the model, an iterator pointing to the row in the model, and, optionally, an argument for user data. The function is tasked with setting the appropriate attributes of the cell renderer. For example, this callback would format floating point numbers:

```
func <- function(column, cell_renderer, model, iter, data) {
  val <- model$getValue(iter, 0)$value
  f_val <- sprintf("%.3f", val)
  cell_renderer['text'] <- f_val
  cell_renderer['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))))
cell_renderer <- gtkCellRendererText()
view$insertColumnWithAttributes(0, "numbers", cell_renderer,
                                text = 0)

column <- view$getColumn(0)
column$setCellDataFunc(cell_renderer, func)
```

The last line is the key: calling `setCellDataFunc` registers our custom-formatting function with the view column.

One drawback of 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 in the general case beyond `GtkTreeView`, we could implement a new type of `GtkCellRenderer`. See Chapter 11 for more details on extending GTK+ classes.

9.2 Displaying 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 nonhierarchical 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 9.6: Defining a tree

Below, we create a tree based on the `Cars93` data set, where the car models (`Model`) are organized by manufacturer (`Manufacturer`), i.e., each model row is the child of its manufacturer row:

```
model <- gtkTreeStore("gchararray")
by(Cars93, Cars93$Manufacturer, function(DF) {
  parent_iter <- model$append()
  model$setValue(parent_iter$iter, column = 0, value =
    DF$Manufacturer[1])
  sapply(DF$Model, function(car_model) {
    child_iter <- model$append(parent = parent_iter$iter)
    if (is.null(child_iter$retval))
      model$setValue(child_iter$iter, column = 0,
        value = car_model)
  })
})
```

To retrieve a value from the tree store using its path we have:

```
iter <- model$getIterFromString("0:0")
model$getValue(iter$iter, column = 0)$value
```

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

As shown in the above 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 a 0-based column index. An entire row can be assigned through the `set` method. The method uses positional arguments to specify the column and the value, in alternating fashion. 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 use of `Gtk-TreeIter`, introduced previously in the context of flat tables. Here we perform a depth-first traversal of our `Cars93` model to obtain the model values:

```
iter <- model$getIterFirst()
values <- NULL
while(iter$retval) {
  child_iter <- model$iterChildren(iter$iter)
  while(child_iter$retval) {
    values <- c(values, model$get(child_iter$iter, 0)[[1]])
    child_iter$retval <- model$iterNext(child_iter$iter)
  }
  iter$retval <- model$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.

Row manipulations Rows within a store can be rearranged using several methods. Call the `swap` method 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 can be removed using the `remove` method. To remove every row, call the `clear` method.

Displaying data as a tree

Once a hierarchical data set 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 trigger the emission of the signals `row-expanded` and `row-collapsed`, respectively.

Example 9.7: 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 9.6 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.

Creating a basic view is similar to that for rectangular data already presented:

```
view <- gtkTreeView()  
view$insertColumnWithAttributes(0, "Make",  
                                gtkCellRendererText(), text = 0)
```

```
[1] 1
```

```
view$setModel(model)
```

To demonstrate that `GtkTreeView` supports both hierarchical and flat tabular models, we will create an analogous `RGtkDataFrame` and set it on the view:

```
model <- rGtkDataFrame(Cars93[, "Model", drop=FALSE])  
view$setModel(model)
```

9.3 Model-based combo boxes

Basic combo-box usage was discussed in Section 8.3; 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, we should pass the model to the constructor `gtkComboBox`. This model can be changed or set through the `setModel` method and is returned by `getModel`. 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.

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` methods behave analogously with 0-based indices.

Example 9.8: A combo box with memory

This example uses an editable combo box as a simple interface to the R help system. Along the way, we record the number of times the user searches for a page.

Our model for the combo box will be an `RGtkDataFrame` instance with three columns: a function name, a string describing the number of visits, and an integer to record the number of visits.

```
model <- rGtkDataFrame(data.frame(filename = character(0),
                                visits = character(0),
                                nvisits = integer(0),
                                stringsAsFactors = FALSE))
```

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, we 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. We create the combo box with this model using the first column for the text:

```
combo_box <- gtkComboBoxEntryNewWithModel(model,
                                           text.column = 0)
```

It is not currently possible to put tooltip information on the drop-down elements of a combo box, as was done with a treeview. Instead, we borrow from popular web browser interfaces and add textual information about the number of visits to the drop-down menu. This requires us to pack in a new cell renderer to accompany the original label provided by the `gtkComboBoxEntry` widget:

```
cell_renderer <- gtkCellRendererText()
combo_box$packStart(cell_renderer)
combo_box$addAttribute(cell_renderer, "text", 1)
cell_renderer['foreground'] <- "gray50"
cell_renderer['ellipsize'] <- "end"
cell_renderer['style'] <- "italic"
cell_renderer['alignment'] <- "right"
```

This helper function will be called each time a help page is requested. It first updates the visit information, selects the text for easier editing the next time around, then calls `help`.

```
callHelpFunction <- function(combo_box, value) {
  model <- combo_box$getModel()
  ind <- match(value, model[,1,drop=TRUE])
  nvisits <- model[ind, "nvisits"] <- model[ind, "nvisits"]+1
```



```
model[ind, "visits"] <-  
  sprintf(ngettext(nvisits, "%s visit", "%s visits"), nvisits)  
## select for easier editing  
combo_box$getChild()$selectRegion(start = 0, end = -1)  
help(value)  
}  
gSignalConnect(combo_box, "changed",  
  f = function(combo_box, ...) {  
    if(combo_box$getActive() >= 0) {  
      value <- combo_box$getActiveText()  
      callHelpFunction(combo_box, value)  
    }  
  })
```

When the user enters a new value in the entry, we need to check whether we have previously accessed the item. If not, we add the value to our model.

```
gSignalConnect(combo_box$getChild(), "activate",  
  f = function(combo_box, entry, ...) {  
    value <- entry$getText()  
    if(!any(value == combo_box$getModel()[,1])) {  
      model <- combo_box$getModel()  
      tmp <- data.frame(filename = value, visits = "",  
                        nvisits = 0,  
                        stringsAsFactors = FALSE)  
      model$appendRows(tmp)  
    }  
    callHelpFunction(combo_box, value)  
  }, data = combo_box, user.data.first = TRUE)
```

We place this in a minimal GUI with a label:

```
window <- gtkWindow(show = FALSE)  
window['border-width'] <- 15  
hbox <- gtkHBox(); window$add(hbox)  
hbox$packStart(gtkLabel("Help on:"))  
hbox$packStart(combo_box, expand = TRUE, fill = TRUE)  
#  
window$show()
```

An alternative approach would be to use the completion support of `GtkEntry`, presented next, but we leave that as an exercise to the reader.

9.4 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 automatically add the common prefix to the entry widget; `popup-single-match` is a logical indicating whether a pop up 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 (lowercase), 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 9.9: Text entry with completion

This example illustrates the steps to add completion to a text entry.

We create an entry with a completion database:

```
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:

```
model <- rGtkDataFrame(state.name)
completion$setModel(model)
completion$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, so we define our own match function:

```
matchAnywhere <- function(completion, key, iter, user.data) {
  model <- completion$getModel()
  row_value <- model$getValue(iter, 0)$value
  key <- completion$getEntry()$getText() # case sensitivity
  grepl(key, row_value)
}
completion$setMatchFunc(matchAnywhere)
```

We get the string from the entry widget, not the passed value, as the latter has been standardized to lowercase.

9.5 Sharing buffers between text entries

As of GTK+ version 2.18, multiple instances of `GtkEntry` can synchronize their text through a shared buffer. Each entry obtains its text from the same underlying model, a `GtkEntryBuffer`. Here, we construct two entries, with a shared buffer:

```
buffer <- gtkEntryBuffer()
entry1 <- gtkEntry(buffer = buffer)
entry2 <- gtkEntry(buffer = buffer)
entry1$setText("echo")
entry2$getText()
```

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

The change of text in "entry1" has been reflected in "entry2".

9.6 Text views

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 can be set with the `setBuffer` method. Text views provide native scrolling support and thus are easily added to a scrolled window (Section 7.4).

Example 9.10: Basic `gtkTextView` usage

The steps to construct a text view consist of:

```
view <- gtkTextView()
scrolled_window <- gtkScrolledWindow()
scrolled_window$add(view)
scrolled_window$setPolicy("automatic", "automatic")
##
window <- gtkWindow()
window['border-width'] <- 15
window$add(scrolled_window)
```

To set all the text in the buffer requires accessing the underlying buffer:

```
buffer <- view$getBuffer()
buffer$setText("Lorem ipsum dolor sit amet ...")
```

Manipulating the text requires an understanding of how positions are referred to within the buffer (iterators or marks). As an indicator, to get the contents of the buffer can be done as follows:

```
start <- buffer$getStartIter()$iter
end <- buffer$getEndIter()$iter
buffer$getText(start, end)
```

```
[1] "Lorem ipsum dolor sit amet ..."
```

Adding text Text can be added programmatically through various methods of the text buffer. The most basic `setText`, which simply replaces the current text, is shown in the example above. 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.

Properties By default, the text in a view is editable. This can be disabled through the `editable` property. Typically, we then set the `cursor-visible` property to `"FALSE"` so that the cursor is hidden:

```
view['editable'] <- FALSE
view['cursor-visible'] <- FALSE
```

Formatting The text view supports several general formatting options. Automatic line wrapping is enabled through `setWrapMode`, which takes a value from `GtkWrapMode`: one of `"none"`, `"char"`, `"word"`, and `"word_char"`. The justification for the entire buffer is controlled by the `justification` property, which takes a `GtkJustification` value from `"left"`, `"right"`, `"center"`, and `"fill"`. The global value may be overridden for parts of the text buffer through the use of text tags (see Section 9.7). The left and right margins are adjusted through the `left-margin` and `right-margin` properties.

Fonts The size and font can be set globally for a text view using the `modifyFont` method. To set the font for specific regions, use text tags (see Section 9.7). The font is specified as a Pango font description, which may be generated from a string through `pangoFontDescriptionFromString`. These strings may contain up to three 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 unit “px” is 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.

9.7 Text buffers

Text buffer properties include `text` for the stored text and `has-selection` to indicate whether text is currently selected in a view. The buffer also tracks whether it has been modified. This information is available through the buffer `getModified` method, which returns `TRUE` if the buffer has changed. To clear this state, such as when a buffer has been saved to disk, we can pass `FALSE` to `setModified`.

In order to do more with a text buffer, such as retrieve a selection, or modify text attributes, we need to become familiar with the two mechanisms for referencing text in a buffer: iterators and marks. A text iterator is an opaque, transient pointer to a region of text, whereas a text mark specifies a location that remains valid across buffer modifications.

Iterators

In GTK+ a *text iterator* is the primary means of specifying a position in a buffer. As mentioned in Section 9.1, iterators are typically transient, in the sense that they are invalidated or updated by reference when their source is modified.

Several methods of the text buffer return iterators marking positions in the buffer. Iterators are returned as lists with two components: `iter`, which represents the actual C iterator object, and `retval`, a logical value indicating whether the iterator is valid. The beginning and end of the buffer are returned by the methods `getStartIter` and `getEndIter`. Both of these iterators are returned together in a list by the method `getBounds`. For example:

```
bounds <- buffer$getBounds()  
bounds
```

```
$retval  
NULL
```

```

$start
<pointer: 0x116458ec0>
attr("interfaces")
character(0)
attr("class")
[1] "GtkTextIter" "GBoxed"      "RGtkObject"

$end
<pointer: 0x116458ce0>
attr("interfaces")
character(0)
attr("class")
[1] "GtkTextIter" "GBoxed"      "RGtkObject"

```

The current selection is returned by the method `getSelectionBounds`, as a list of the same structure. If there is no selection, then `retval=FALSE`.

We can also obtain an iterator for a specific position in a document. The method `getIterAtLine` will return an iterator pointing to the start of the line, which is specified by a 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. For example, we might request the seventh character of the first line:

```

iter <- buffer$getIterAtLineOffset(0, 6)
iter$iter$getChar() # unicode, not text

```

```
[1] 105
```

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.

Once we obtain an iterator, we typically enter a loop that performs some operation on the text at the iterator position and updates the iterator with each iteration. This requires retrieving the text to which an iterator refers. The character at the iterator position is returned by `getChar`. We obtain the first character in the buffer:

```

bounds$start$getChar() # unicode

```

```
[1] 76
```

To obtain the text between two text iterators, call the `getText` method on the left iterator, passing the right iterator as an argument:

```

bounds$start$getText(bounds$end)

```

```
[1] "Lorem ipsum dolor sit amet ..."
```

The `insert` method will insert text at a specified iterator:

```
buffer$insert(bounds$start, "prefix")
```

The `delete` method will delete the text between two iterators. An important observation is that we always pass the actual iterator, i.e., the `iter` component of the list, to the above methods. Passing the original list would not work.

Next, we introduce the methods for updating an iterator. We can move an iterator forward or backward, stopping at a certain type of landmark. Supported landmarks include characters (`forwardChar`, `forwardChars`, `backwardChar`, and `backwardChars`), words (`forwardWordEnd` and `backwardWordStart`), and sentences (`backwardSentenceStart` and `forwardSentenceEnd`). There are also various methods, such as `insideWord`, for determining the textual context of the iterator. Example 9.11 shows how some of the above are used, in particular how these methods update the iterator rather than return a new one.

Example 9.11: Finding the word that is clicked by the user

This example shows how we can find the iterator corresponding to a mouse click. In the callback we obtain the *X* and *Y* coordinates of the mouse-button-press event, find the corresponding iterator, and retrieve the surrounding word:

```
gSignalConnect(view, "button-press-event",
  f=function(view, event, ...) {
    start <- view$getIterAtLocation(event$getX(),
                                   event$getY())$iter

    end <- start$copy()
    start$backwardWordStart()
    end$forwardWordEnd()
    val <- start$getText(end)
    print(val)
    return(FALSE) # call next handler
  })
```

Marks

A text mark tracks a position in the document that is relative to other text and is preserved across buffer modifications. We can think of a mark as an invisible object stuck between two characters. An example is the text cursor, the position of which is represented by a mark.

Marks are identified by name. We retrieve the mark for the cursor, which is called "insert":

```
insert <- buffer$getMark("insert")
```

To access the text at a mark, we need the corresponding iterator:

```
insert_iter <- buffer$getIterAtMark(insert)$iter
bounds$start$getText(insert_iter)
```

```
[1] "Lorem ipsum dolor sit amet ..."
```

Marks have a gravity of "left" or "right", with "right" being the default. If text is inserted at a mark with right gravity, then the mark is moved to the end of the insertion. A mark with left gravity would not be moved. This is intuitive if we relate it to the behavior of the text cursor, which has right gravity. For obvious reasons, the cursor always advances as the user inserts text by typing. We demonstrate this programmatically:

```
insert_iter$getOffset()
```

```
[1] 36
```

```
buffer$insert(insert_iter, "at insertion point")
buffer$getIterAtMark(insert)$iter$getOffset()
```

```
[1] 54
```

A custom mark is created with its name, gravity, and position. We create one for the start of the document:

```
mark <- buffer$createMark(mark.name = "start",
                           where = buffer$getStartIter()$iter,
                           left.gravity = TRUE)
```

If we set `left.gravity` to "TRUE", the iterator will not move when text is inserted.

Tags

Tags are annotations placed on specific regions of a text buffer. To create a tag, we call the `createTag` method, which takes an argument for each attribute to apply to the text. Here, we create three tags: one for bold text, one for italicized text, and one for large text:

```
tag_bold <- buffer$createTag(tag.name="bold",
                             weight=PangoWeight["bold"])
tag_emph <- buffer$createTag(tag.name="emph",
                             style=PangoStyle["italic"])
tag_large <- buffer$createTag(tag.name="large",
                              font="Serif normal 18")
```

Next, we associate the tags with one or more regions of text:


```
iter <- buffer$getBounds()
buffer$applyTag(tag_bold, iter$start, iter$end) # iters update
buffer$applyTagByName("emph", iter$start, iter$end)
```

Selection and the clipboard

The selection is defined by the text buffer as the region between the "insert" and "selection_bound" marks. While we could move the marks around directly, calling `selectRange` is more efficient and convenient. Here, we select the first word:

```
start_iter <- buffer$getStartIter()$iter
end_iter <- start_iter$copy(); end_iter$forwardWordEnd()
buffer$selectRange(start_iter, end_iter)
```

`GtkTextBuffer` provides some convenience methods for interaction with the clipboard: `copyClipboard`, `cutClipboard`, and `pasteClipboard`. To use these, we first need a clipboard object:

```
clipboard <- gtkClipboardGet()
```

We can then, for example, copy the selected text (the first word) and paste it at the end:

```
buffer$copyClipboard(clipboard)
buffer$pasteClipboard(clipboard,
  override.location = buffer$getEndIter()$iter,
  default.editable = TRUE)
```

The `default.editable` argument indicates that the pasted text should be editable. If we had passed `NULL` to the `override.location` argument, the insertion would have occurred at the cursor.

Inserting nontext items

If desired, we can insert images and/or widgets into a text buffer. The method `insertPixbuf` will insert a `GdkPixbuf` object. The buffer will count the image as a character, although `getText` will obviously not return the image.

Arbitrary child widgets, like buttons, can also be inserted. First, we must create an anchor in the text buffer with `createChildAnchor`:

```
anchor <- buffer$createChildAnchor(buffer$getEndIter()$iter)
```

To add the widget, we call the textview method `addChildAtAnchor`:

```
button <- gtkButton("click me")
view$addChildAtAnchor(button, anchor)
```

```

> 2 + 2
[1] 4
> lm(wt ~ mpg, data=mtcars)

Call:
lm(formula = wt ~ mpg, data = mtcars)

Coefficients:
(Intercept)      mpg
   6.0473      -0.1409
>

```

Figure 9.7: A basic R terminal implemented using a `gtkTextView` widget.

Example 9.12: A simple command-line interface

This example shows how to create a simple command-line interface with the `textview` widget (Figure 9.7). While few statistical applications will include a command-line widget, the example is familiar and shows several different, but useful, aspects of the widget.

We begin by defining our `textview` widget and retrieving its buffer. Then we specify a fixed-width font for the buffer:

```

view <- gtkTextView()
buffer <- view$getBuffer()
font <- pangoFontDescriptionFromString("Monospace")
view$modifyFont(font) # widget wide

```

We will use a few formatting tags, defined next. We do not need the tag objects as variables in the workspace, as we refer to them later by name.

```

buffer$createTag(tag.name = "cmdInput")
buffer$createTag(tag.name = "cmdOutput",
                 weight = PangoWeight["bold"])
buffer$createTag(tag.name = "cmdError",
                 weight = PangoStyle["italic"], foreground = "red")
buffer$createTag(tag.name = "uneditable", editable = FALSE)

```

We define a mark to indicate the beginning of a newly entered command; and another mark tracks the end of the buffer:

```

start_cmd <- buffer$createMark("start_cmd",
                              buffer$getStartIter()$iter,
                              left.gravity = TRUE)
bufferEnd <- buffer$createMark("bufferEnd",
                              buffer$getEndIter()$iter)

```

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:

```
add_prompt <- function(obj, prompt = c("prompt", "continue"),
                      set_mark = TRUE)
{
  prompt <- match.arg(prompt)
  prompt <- getOption(prompt)

  end_iter <- obj$getEndIter()
  obj$insert(end_iter$iter, prompt)
  if(set_mark)
    obj$moveMarkByName("start_cmd", end_iter$iter)
  obj$applyTagByName("uneditable", obj$getStartIter()$iter,
                    end_iter$iter)
}
add_prompt(buffer) ## place an initial prompt
```

This helper method writes the output of a command to the text buffer:

```
add_output <- function(obj, output, tag_name = "cmdOutput") {
  end_iter <- obj$getEndIter()
  if(length(output) > 0)
    sapply(output, function(i) {
      obj$insertWithTagsByName(end_iter$iter, i, tag_name)
      obj$insert(end_iter$iter, "\n", len=-1)
    })
}
```

We did not arrange to truncate large outputs, but that would be a nice addition. By passing in the tag name, we can specify whether this is normal output or an error message.

This next function uses the `start_cmd` mark and the end of the buffer to extract the current command. The "regex" is used to parse multiline commands.

```
find_cmd <- function(obj) {
  end_iter <- obj$getEndIter()
  start_iter <- obj$getIterAtMark(start_cmd)
  cmd <- obj$getText(start_iter$iter, end_iter$iter, TRUE)
  regex <- paste("\n[", getOption("continue"), "]" , sep = "")
  cmd <- unlist(strsplit(cmd, regex))
  cmd
}
```

The following function takes the current command and evaluates it using the `evaluate` package. It uses a hack (involving `grep1`) to distinguish between an incomplete command and a true syntax error.

```
require(evaluate)
eval_cmd <- function(view, cmd) {
  buffer <- view$getBuffer()
  out <- try(evaluate:::evaluate(cmd, .GlobalEnv),
```

```

        silent = TRUE)

if(inherits(out, "try-error")) {
  ## parse error
  add_output(buffer, out, "cmdError")
} else if(inherits(out[[2]], "error")) {
  if(grepl("end", out[[2]])) {          # a hack here
    add_prompt(buffer, "continue", set_mark = FALSE)
    return()
  } else {
    add_output(buffer, out[[2]]$message, "cmdError")
  }
} else {
  add_output(buffer, out[[2]], "cmdOutput")
}
add_prompt(buffer, "prompt", set_mark = TRUE)
}

```

We arrange that the `eval_cmd` command is called when the return key is pressed next. Other key bindings might also be of interest, such as one for tab completion.

```

gSignalConnect(view, "key-release-event",
               f=function(view, event) {
                 buffer <- view$getBuffer()
                 keyval <- event$getKeyval()
                 if(keyval == GDK_Return) {
                   cmd <- find_cmd(buffer)
                   if(length(cmd) && nchar(cmd) > 0)
                     eval_cmd(view, cmd)
                 }
               })

```

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:

```

scroll_viewport <- function(view, ...) {
  view$scrollToMark(bufferEnd, within.margin = 0)
  return(FALSE)
}
gSignalConnect(buffer, "changed", scroll_viewport, data=view,
               after = TRUE, user.data.first = TRUE)

```

This page intentionally left blank

RGtk2: Application Windows

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. Menu bars and toolbars are the two most common widgets for organizing application-wide actions. An application also needs to report its status in an unobtrusive way. Thus, a typical application window contains, from top to bottom, a menu bar, a toolbar, a large application-specific region, and a status bar. In this chapter, we will introduce actions, menus, toolbars and status bars and conclude by explaining the mechanisms in GTK+ for conveniently defining and managing actions and associated widgets in a large application.

10.1 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:

```
action <- gtkAction(name = "ok", label = "_Ok",
                    tooltip = "An OK button", stock.id = "gtk-ok")
```

The constructor takes arguments `name` (to refer programmatically 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(action, "activate",
               f = function(action, data) {
                 print(action$getName())
               })
```

An action plays the role of a data model describing a command, while widgets that implement the `GtkActivatable` interface are the views and



Figure 10.1: An application window mock-up showing a menu bar, toolbar, and info bar.

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 `setRelatedAction`:

```
button <- gtkButton()
button$setRelatedAction(action)
```

Certain aspects of a proxy widget are coordinated through the action. These include sensitivity and visibility, corresponding to the `sensitive` and `visible` properties. By default, aesthetic properties such as the label and `stock-id` are also inherited.

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(action, "<control>0")
```

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 checkboxes and radio buttons, respectively. Here, we create a toggle action for full-screen mode:

```
full_screen_act <-
```

```

gtkToggleAction("fullscreen", "Full screen",
               "Toggle full screen",
               stock.id = "gtk-fullscreen")
gSignalConnect(full_screen_act, "toggled", function(action) {
  if(full_screen_action['active'])
    window$fullscreen()
  else
    window$unfullscreen()
})

```

We connect to the toggled signal to respond to a change in the action state.

10.2 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 “pop up” when the user interacts with the element, such as by clicking the right mouse button. Menu bars and pop-up menus can 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 10.5.

Menu bars

We will first demonstrate the menu bar, leaving the pop-up menu for later. Figure 10.1 shows a realization. The first step 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 may proxy an action. Usually, a menu bar consists of multiple instances of the other type of menu shell: the menu, `GtkMenu`. Here, we create a menu object for our “File” menu:

```
file_menu <- 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:

```
file_item <- gtkMenuItemNewWithMnemonic(label = "_File")
file_item$setSubmenu(file_menu)
```


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(file_item)
```

In addition to `append`, it is also possible to `prepend` and insert menu items into a menu shell. As with any container, we 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_item <- gtkMenuItemNewWithMnemonic("_Open")
```

This item does not have an associated `GtkAction`, so we need to implement its `activate` signal directly:

```
gSignalConnect(open_item, "activate", function(item) {  
  file.show(file.choose())  
})
```

The item is now ready to be added to the file menu:

```
file_menu$append(open_item)
```

It is recommended, however, that we create menu items that proxy an action. This will facilitate, for example, adding an equivalent toolbar item later. We demonstrate with a “Save” action:

```
save_action <-  
  gtkAction("save", "Save", "Save object", "gtk-save")
```

Then the appropriate menu item is generated from the action and added to the file menu:

```
save_item <- save_action$createMenuItem()  
file_menu$append(save_item)
```

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 apart from the rest of the menu:

```
file_menu$append(gtkSeparatorMenuItem())
```

Toggle menu items, i.e., a label next to a checkbox, are also supported. A toggle action will create one implicitly:

```
auto_save_action <- gtkToggleAction("autosave", "Autosave",  
  "Enable autosave")  
auto_save_item <- auto_save_action$createMenuItem()  
file_menu$append(auto_save_item)
```

Finally, we add our menu bar to the top of a window:

```
main_mindow <- gtkWindow()
vbox <- gtkVBox()
main_mindow$add(vbox)
vbox$packStart(menubar, FALSE, FALSE)
```

Pop-up menus

Example 10.1: Pop-up menus

To illustrate pop-up menus, we 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 pop up a menu when clicked with the third (right) mouse button:

```
button <- gtkButton("Click me with right mouse button")
window <- gtkWindow(); window$setTitle("Popup menu example")
window$add(button)
```

This menu will be shown by calling `gtkMenuPopup` in response to the `button-press-event` signal on the button:

```
gSignalConnect(button, "button-press-event",
  f = function(button, event, menu) {
    if(event$getButton() == 3 ||
        (event$getButton() == 1 && # a mac
         event$getState() == GdkModifierType['control-mask']))
      gtkMenuPopup(menu,
                    button = event$getButton(),
                    activate.time = event$getTime())
    return(FALSE)
  }, data = popup)
```

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`).

The above will pop up a menu, but until we bind a callback to the `activate` signal on each item, nothing will happen when a menu item is selected. Below we supply a stub for sake of illustration:

```
sapply(popup$getChildren(), function(child) {
  if(!inherits(child, "GtkSeparatorMenuItem")) # skip these
    gSignalConnect(child, "activate",
      f = function(child, data) message("replace me"))
})
```

We iterate over the children, avoiding the separator.

10.3 Toolbars

Toolbars are like menu bars in that they are containers for activatable items, but toolbars are not hierarchical. Also, their items are usually visible for the lifetime of the application, not upon user interaction. 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, yet toolbars typically represent actions with buttons. The `GtkToolButton` widget implements this common case. Here, we create a tool button for opening a file:

```
open_button <- gtkToolButton(stock.id = "gtk-open")
```

Tool buttons have a number of properties, including `label` and several for icons. 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 can be added to the toolbar with the `add` method:

```
toolbar$add(open_button)
```

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:

```
save_button <- save_action$createToolItem()
toolbar$add(save_button)
```

A tool button is created from the `saveAction` object of 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:

```
full_screen_button <- full_screen_act$createToolItem()
toolbar$add(full_screen_button)
```

A `GtkToggleToolButton` embeds a `GtkToggleButton`, which is depressed whenever its active property is `TRUE`.

As mentioned above, toolbars, unlike menus, are usually visible for the duration of the application. This is desirable, as the actions in a toolbar are among those most commonly performed. However, care must be taken to conserve screen space. The toolbar *style* controls whether the tool items display their icons, their text, or both. The possible settings are in the `GtkToolbarStyle` enumeration. The default value is specified by the global GTK+ style (theme). Here, we override the default to display only images:

```
toolbar$setStyle("icon")
```

For canonical actions like *open* and *save*, icons are usually sufficient. Some actions, however, may require textual explanation. The `is-important` property on the action will request display of the label in a particular tool item, in addition to the icon:

```
full_screen_act["is-important"] <- TRUE
```

Normally, tool items are tightly packed against the left side of the toolbar. Sometimes, a more complex layout is desired. For example, we may wish to place a *help* item against the right side. We can achieve this with an invisible item that expands against its siblings:

```
expander <- gtkSeparatorToolItem()
expander["draw"] <- FALSE
toolbar$add(expander)
toolbar$childSet(expander, expand = TRUE)
```

The dummy item is a separator with its `draw` property set to `FALSE` and its `expand child` property set to `TRUE`. Now we can add the *help* item:

```
help_action <- gtkAction("help", "Help", "Get help", "gtk-help")
toolbar$add(help_action$createToolItem())
```

It is now our responsibility to place the toolbar at the top of the window, under the menu created in the previous section:

```
vbox$packStart(toolbar, FALSE, FALSE)
```

Example 10.2: 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.

```
gdk_color <- gdkColorParse(palette()[1])$color
color_button <- gtkColorButton(gdk_color)
```

The `gtkColorButton` constructor 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) {
  drawing_area <- gtkDrawingArea()
  drawing_area$setSizeRequest(20, 20)
  drawing_area$modifyBg("normal", color)
  image_item <- gtkImageMenuItem(color)
  image_item$setImage(drawing_area)
  image_item
}
color_items <- sapply(palette(), colorMenuItem)
color_menu <- gtkMenu()
for (item in color_items)
  color_menu$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
  color_button$setColor(color)
}
sapply(color_items, gSignalConnect, "activate",
       colorMenuItemActivated)
```

Finally, we place the color button and menu together in the menu tool button:

```
menu_button <- gtkMenuToolButton(color_button, "Color")
menu_button$setMenu(color_menu)
toolbar$add(menu_button)
```

Some applications may offer a large number of actions, where there is no clear subset of actions that are more commonly performed than the

rest. It would be impractical to place a tool item for each action in a static toolbar. GTK+ provides a *tool palette* widget, which leaves the configuration of a multi-row toolbar to the user, as one solution. The tool items are organized into collapsible groups, and the grouping is customizable through drag-and-drop.

`GtkToolPalette` is a container of `GtkToolItemGroup` widgets, each of which is a container of tool items and implements `GtkToolShell`, like `GtkToolbar`. We begin our brief example by creating two groups of tool items:

```
file_group <- gtkToolItemGroup("File")
file_group$add(gtkToolButton(stock.id = "gtk-open"))
file_group$add(save_action$createToolItem())
help_group <- gtkToolItemGroup("Help")
help_group$add(help_action$createToolItem())
```

The groups are then added to an instance of `GtkToolPalette`:

```
palette <- gtkToolPalette()
palette$add(file_group)
palette$add(help_group)
```

Finally, we can programmatically collapse a group:

```
help_group$setCollapsed(TRUE)
```

10.4 Status reporting

Status bars

In GTK+, a status bar is constructed through the `gtkStatusbar` function. Status bars must be placed at the bottom of top-level windows. A status bar keeps various stacks of messages for display. Stacks and messages are associated with a context ID, which represents a message source. It is required to register each context ID against a string description. The visibility depends on the ordering of the global stack, while the context ID allows the status bar to maintain a separate message stack for each part of an application, without worry of interference between components.

To display a message, we push it onto the top of the global stack, as well as a context stack, through the `push` method, which expects an integer value for `context.id` and a message. To pop a message from a context stack, pass the context ID to the `pop` method. If the message was on top of the global stack, the next message down becomes visible.

Below, we create a status bar, register a context for I/O-related messages, display the message, and then pop it to restore the original state:

```
statusbar <- gtkStatusbar()
```

```
io_id <- statusbar$getContextId("I/O")
statusbar$push(io_id, "Incomplete final line")
## ...
statusbar$pop(io_id)
```

Info bars

An *info bar* is similar in purpose to a message dialog, but it is intended to be less obtrusive. Typically, an info bar raises from the bottom of the window, displaying a message, possibly with response buttons. It then fades away after a number of seconds. The focus is not affected, nor is the user interrupted. GTK+ provides the `GtkInfoBar` class for this purpose. The use is similar to a dialog: we place widgets into a content area and listen to the response signal.

We create our info bar:

```
info_bar <- gtkInfoBar(show=FALSE)
info_bar$setNoShowAll(TRUE)
```

We call `setNoShowAll` to prevent the widget from being shown when `showAll` is called on the parent. Normally, an info bar is not shown until it has a message.

We will emit a warning message by adding a simple label with the text and specifying the message type as `warning`, from `GtkMessageType`:

```
label <- gtkLabel("Warning, Warning ....")
info_bar$setMessageType("warning")
info_bar$getContentArea()$add(label)
```

A button to allow the user to hide the bar can be added as follows:

```
info_bar$addButton(button.text = "gtk-ok",
                  response.id = GtkResponseType['ok'])
```

This is similar to the dialog API: the appearance of the “Ok” button is defined by the stock ID `gtk-ok`, and the response ID will be passed to the response signal when the button is clicked. Our handler simply closes the bar:

```
gSignalConnect(info_bar, "response",
               function(info_bar, resp.id) info_bar$hide())
```

Finally, we add the info bar to our main window and show it:

```
vbox$packStart(info_bar, expand = FALSE)
info_bar$show()
```

10.5 Managing a complex user interface

Complex applications implement a large number of actions and operate in a number of different modes. Within a given mode, only a subset of actions are applicable. For example, a word processor may have an editing mode and a print preview mode. GTK+ provides a *user interface manager*, `GtkUIManager`, to manage the layout of the toolbars and menu bars across multiple user-interface modes. We illustrate through an example.

The steps required to use GTK+'s UI manager are:

1. construct the UI manager,
2. specify in XML the layout of the menu bars and toolbars,
3. define the actions in groups,
4. connect the action group to the UI manager,
5. set up an accelerator group for keyboard shortcuts, and finally
6. display the widgets.

Example 10.3: UI manager example

In this example, we show how to use a UI manager to create the menu and toolbars for a data-frame editor, similar to, but with enhanced functionality, as produced on some platforms by the `data.entry` function.

Our menu bar and toolbar layout is expressed in XML according to a schema specified by the UI manager framework. The XML can be stored in a file or an R character vector. The structure of the file can be grasped quickly from this example:

```
ui.xml <- readLines(out <- textConnection('
<ui>
  <menubar name="menubar">
    <menu name="FileMenu" action="File">
      <menuitem action="Save" />
      <menuitem action="SaveAs" />
      <menu name="Export" action="Export">
        <menuitem action="ExportToCSV" />
        <menuitem action="ExportToSaveFile" />
      </menu>
      <separator />
      <menuitem name="FileQuit" action="CloseWindow" />
    </menu>
    <menu action="Edit">
      <menuitem name="EditUndo" action="Undo" />
      <menuitem name="EditRedo" action="Redo" />
    </menu>
  </menubar>
</ui>')
```

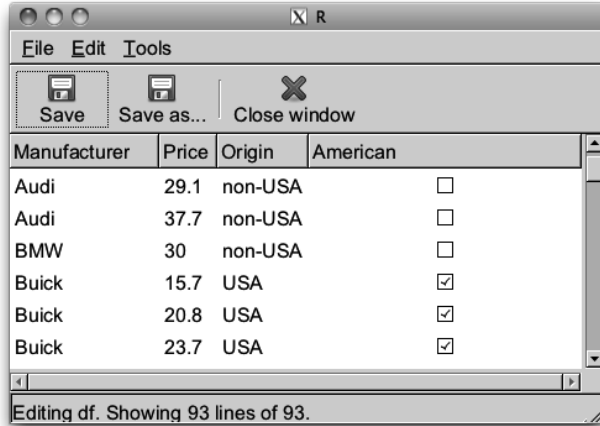



Figure 10.2: An instance of an editable data frame with menu and toolbar specified using an instance of `GtkUIManager`. This example, implements the command pattern to provide simple undo and redo functionality.

```

    <menuitem action="ChangeColumnName" />
  </menu>
  <menu action="Tools">
    <menuitem action="Filter" />
    <menuitem action="Sort" />
  </menu>
</menubar>
<toolbar name="toolbar">
  <toolitem action="Save"/>
  <toolitem action="SaveAs"/>
  <separator />
  <toolitem action="CloseWindow"/>
</toolbar>
</ui>'), warn=FALSE)
close(out)

```

We used indenting to show the nesting of the menus. For menus we see the use of menu bars, menus and menu items. The menu and menu items have a corresponding action associated with them, which can provide a callback.

If `uimanager` is our `GtkUIManager` instance, then we can add this through the command:

```
id <- uimanager$addUiFromString(ui.xml)
```

Alternately, we could load the code from a file. The return value is an ID that can be used to unmerge this part of the UI. The ability to merge and unmerge parts of the UI is one main attraction for using this framework, although we do not illustrate that here.

To define the actions, we can use lists. Each item contains six pieces of information: a name (which we use in fun to call the appropriate method), a stock-id, a label, a keyboard accelerator, a tooltip, and finally a callback for when the action is invoked. This list defines the file menu:

```
file_list <-
  list(## name, ID, label, accelerator, tooltip, callback
    list("File", NULL, "_File", NULL, NULL, NULL),
    list("Save", "gtk-save", "Save", "<ctrl>S",
        "Save data to variable", fun),
    list("SaveAs", "gtk-save", "Save as...", NULL,
        "Save data to variable", fun),
    list("Export", NULL, "Export", NULL, NULL, NULL),
    list("ExportToCSV", "gtk-export", "Export to CSV",
        NULL, "Save data to CSV file", fun),
    list("ExportToSaveFile", "gtk-export",
        "Export to save file", NULL,
        "Save data to save() file", fun),
    list("CloseWindow", "gtk-close", "Close window",
        "<ctrl>W", "Close current window", fun)
  )
```

We can add these items to an action group, along the lines of

```
action_group <- gtkActionGroup("FileGroup")
action_group$addActions(file_list)
```

We can then insert the action group into the UI manager:

```
uimanager$insertActionGroup(action_group, 0)
```

The position (0) is used to determine which action will be called, when there is more than one with the same name.

We now place the UI manager controls into the GUI. The uimanager instance creates widgets that can be retrieved through its `getWidget` method. The following code uses this to sketch out the layout of the GUI:

```
window <- gtkWindow(show = FALSE)
##
vbox <- gtkVBox()
window$add(vbox)
##
menubar <- uimanager$getWidget("/menubar")
vbox$packStart(menubar, FALSE)
toolbar <- uimanager$getWidget("/toolbar")
vbox$packStart(toolbar, FALSE)
```

```
## ...
```

The menubar and toolbar widgets are referred to by their path, which comes from the names specified in the XML description separated by `"/`. So, in the definition above, the following lines define the path `"/menubar"`, where `<ui>` is always the root element, and may be omitted from the path:

```
<ui>
  <menubar name="menubar">
  ...
```

Finally, to connect the UI manager to the window, we add the keyboard accelerator group:

```
window$addAccelGroup(uimanager$getAccelGroup())
```

Figure 10.2 shows an illustration of the finished application. The full details are found in the code in our accompanying package `ProgGUIinR`.

Command pattern Now, we discuss how the command pattern is implemented to provide a simple undo and redo feature to our editing. According to *Head First Design Patterns*^[7], the command pattern is used to encapsulate a request (method call) as an object. A basic command object has just one method, `execute`. Any needed parameters are stored in the object as properties. The command pattern has GUI-related applications beyond the undo and redo stack, including the action objects (i.e., instances of `GtkAction`) that are managed by `GtkUIManager`.

For our implementation of the undo/redo stack, we use a reference class with fields:

```
Command <- setRefClass("Command",
  fields = list(
    receiver="ANY",
    meth="character",
    params="list",
    old_params="list"
  ))
```

The receiver property stores the object referred to in the method call. For a simple function call, this could be the environment enclosing the function. The `meth` property is the name of the method, and `params` is a list of parameters. With these we define the main methods:

```
Command$methods(
  initialize = function(receiver, meth, ...) {
```

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

```

    .params <- list(...)
    initFields(receiver = receiver, meth = meth,
              params = .params, old_params = .params)
    callSuper()
  },
  execute = function(params) {
    do.call(call_meth(meth, receiver), params)
  })

```

Notice we pass in the arguments to our execute method, rather than use those in the property params. This allows us to implement the do and undo methods in a similar manner:

```

Command$methods(
  do = function() {
    out <- execute(params)
    old_params$value <<- out
  },
  undo = function() execute(old_params)
)

```

This assumes the method executed can return a value that can be used to reverse the call. If a method call is not so straightforward to reverse, we need only subclass the Command call and provide a new undo method.

A simple illustration might be:

```

x <- 1
set_x <- function(value) {
  old <- x
  x <<- value
  old
}
cmd <- Command$new(.GlobalEnv, "set_x", value = 2)
cmd$do(); x

```

```
[1] 2
```

```
cmd$undo();
```

```
x
```

```
[1] 1
```

In our example, we create a stack of commands to keep track of what was done. This stack has methods add, undo, and redo, each calling the do or undo method of the appropriate command in the stack.

The first command we add to the stack is the setting of a column name on a data frame:

```
cmd <- Command$new(df_model, "set_col_name", j=j, value=value)
command_stack$add(cmd)
```

To explain, `df_model` is an instance of a yet-to-be-defined reference class defining a data model for the data frame being edited, and `j` and `value` are determined by a dialog called before the command is created. The point is, the method call for the `df_model` object is encapsulated along with the needed parameters (a column number and new name) and then added to the command stack. The `add` method calls the `do` method of the command to invoke the changing of the name.

The data frame model (defined in our reference class `DfModel`) is a wrapper around an `RGtkDataFrame` object that holds the data. The method call above is implemented by:

```
DfModel$methods(
    get_col_name = function(j) varnames[j,1],
    get_col_names = function() varnames[,1],
    set_col_name = function(j, value) {
        "Set name, return old"
        old_col_name <- get_col_name(j)
        varnames[j,1] <<- value
        old_col_name
    })
```

We return the old value, as that is required by the implementation of the `do` method for the commands. An instance of `RGtkDataFrame` stores the variable (column) names, hence the double index. This allows us to listen for changes through the row-changed signal on the model. The details, and more, are in the accompanying package.

Extending GObject Classes

GTK+, as well as several of its dependencies, with the notable exception of Cairo, is based on the GObject library for object-oriented programming in C. GObject forms the basis of many other open-source projects, including the GNOME and XFCE desktops and the GStreamer multimedia framework.

Given the broad use of signals in the GTK+ API, it is very rarely necessary to extend a widget class when developing a typical GUI. However, it is generally good practice to encapsulate the behavior of a widget in a formal class. Although there are several such formalisms in R, RGtk2 provides one that is congruent with the rest of GTK+. It interfaces with parts of GObject and permits the R programmer to create new GObject classes in R. A subclass can override certain methods inherited from its parent and define new methods, properties, and signals. If a method is declared by a C class, it can be overridden only if it is a so-called *virtual* method, and there is no documentation as to which methods are virtual. There is a loose convention that every signal has a corresponding virtual method. The ultimate resource is the C header files. A bug in a method override could very easily crash R, so use of this feature takes some commitment from the programmer. Any method declared by an R class may be overridden by an R subclass.

Our example will be a GUI that displays a scatterplot along with a slider for adjusting the alpha level of the points (Figure 11.1). Usually, a slider operates in linear fashion. When there are a large number of points, on the order of tens of thousands or more, changing the alpha level does not have a strong visual effect until it approaches its lower limit. We desire greater control in the lower part of the alpha scale, without limiting the range of the slider. To achieve this, we need to perform a nonlinear transformation from the slider value to the alpha of the plot and reflect that transformation in the label on the slider. One solution is to connect to the `format-value` signal to override the text in the label. We present an alternative that involves extending `GtkHScale` and overriding its `format_value` virtual method.

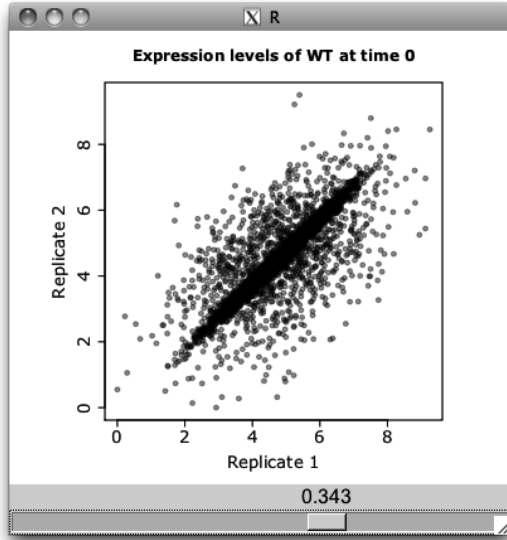


Figure 11.1: An interface using a custom slider to adjust alpha levels in a nonlinear manner.

A class is defined by calling `gClass`, to which is passed the class name, the name of the parent class and a number of list arguments that define the properties, signals and methods of the class. For the sake of cleanliness, everything is defined as part of the `gClass` call:

```
tform_scale_type <-
  gClass("RTransformedHScale", "GtkHScale",
    .props = list(
      gParamSpec(type = "R", name = "expr", nick = "e",
        blurb = "Transformation of scale value",
        default.value = expression(x))
    ),
    GtkScale = list(
      format_value = function(self, x)
        as.character(self$transformValue(x))
    ),
    .public = list(
      getExpr = function(self) self["expr"],
      getTransformedValue = function(self)
        self$transformValue(self$value)
    ),
    .private = list(
      transformValue = function(self, x)
        eval(self$expr, list(x = x))
    )
  )
```

```
)
)
```

The class definition for `RTransformedHScale` starts with a property for the R expression that transforms the value from the slider to the alpha level. A property is defined by a `GParamSpec` structure that specifies a name, nickname, descriptive blurb, value type, and other options. There are subclasses of `GParamSpec` for particular types that permit specification of further constraints. For example, `GParamSpecInt` is specific to integers and can be configured to restrict its valid range of integer values between a minimum and maximum. Many `GParamSpec` subclasses also permit default values. The type argument may refer to any C type by name. The names of R types, like “integer” and “character”, are mapped to the corresponding scalar C type, if available. An “R” property, like our expression, stores any native R value. The actual R type, as returned by `typeof`, may be specified as the `s.type` argument; otherwise, it is taken from the default value.

We turn our attention to the methods in the class definition. The class overrides the `format_value` virtual from `GtkScale` and defines two public methods, `getExpr` and `getTransformedValue`, for retrieving the transformation expression and the transformed value, respectively. There is one private method, `transformValue`, that is a utility for evaluating the expression on the current value.

Methods are implemented with R functions that are grouped into lists. The names of the list identify the methods. An override is placed into the list corresponding to the class in which the original method is declared. For new methods, the division is by the access level: `public`, `protected`, or `private`. Public members can be accessed by any code, while protected members are restricted to methods belonging to the same class or a subclass. Access to private members is the most restricted, as they are available only to methods in the same class.

A function implementing a virtual method may delegate to the method that it overrides. This is achieved by calling the `parentHandler` function and passing it the name of the method and the arguments to forward to the method. This is similar to the `super` function in `qtbase`. For example, in the override of `format_value` in the `RGtkTransformedHScale` class, we could call `parentHandler("format_value", self, x)` to delegate to the implementation of `format_value` in `GtkScale`.

If a non-function, like a vector, is placed in the `.public`, `.protected`, or `.private` list, it represents a field, which is initialized to the given value.

Two elements of the class definition that are not in the example above are the list of signal definitions and the initialization function. The signal definition list is passed as a parameter named `.signals` and contains a list for each signal. Each list includes the name, return type, and parameter types of the signal. The types can be specified in the same format as

used for property definitions. The initialization function, passed as the `.initialize` parameter, is invoked whenever an instance of the class is created, before any properties are set. It takes the newly created instance of the class as its only parameter.

The next step in our example is to create an instance of `RGtkTransformedHScale` and to register a handler on the `value-changed` signal that will draw the plot using the transformed value as the alpha setting:

```
adj <- gtkAdjustment(0.5, 0.15, 1.00, 0.05, 0.5, 0)
s <- gObject(tform_scale_type, adjustment = adj,
             expr = expression(x^3))
gSignalConnect(s, "value_changed", function(scale) {
  plot(ma_data, col = rgb(0,0,0, scale$getTransformedValue()),
       xlab = "Replicate 1", ylab = "Replicate 2",
       main = "Expression levels of WT at time 0", pch = 19)
})
```

Instances of any `GObject` class can be created using the `gObject` function. The value of the `expr` property is set to the R expression x^3 when the object is created. The signal handler now calls the new `getTransformedValue` method, instead of `getValue` as in the original version. The `ma_data` object is a matrix of points that is meant to resemble expression values from two replicates of a microarray experiment.

We complete the example by placing the slider and a graphics device in a window:

```
win <- gtkWindow(show = FALSE)
da <- gtkDrawingArea()
vbox <- gtkVBox()
vbox$packStart(da)
vbox$packStart(s, FALSE)
win$add(vbox)
win$setDefaultSize(400, 400)
#
require(cairoDevice)
asCairoDevice(da)
#
win$showAll()
par(pty = "s")
s$setValue(0.7)
```

Part III

The qtbase Package

This page intentionally left blank

12.1 The Qt library

Qt is an open-source, 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., 2-D 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. It is now owned by Nokia, a major cell-phone manufacturer. While it was originally unavailable as open source on every platform, version 4 was released universally under the GPL. With the release of Qt 4.5, Nokia additionally placed Qt under the LGPL, so it is available for use in proprietary software as well. Popular software developed with Qt includes the communication application Skype and the KDE desktop for Linux. The desktop version of RStudio uses the `QWebView` widget to present a cross-platform web application on the desktop. This book assumes version Qt 4.7.3 and should remain compatible for the remainder of the 4.x series.

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 `qtbase` package interfaces with every module of the library. As its name suggests, `qtbase` forms the base for a number of R packages that provide high-level special-purpose interfaces to Qt. The `qtpaint` package extends the `QGraphicsView` canvas to better support interactive statistical graphics. Features include: a layered buffering strategy, efficient spatial queries for mapping user actions to the data, and an OpenGL renderer optimized for statistical plots. An interface resembling that of the `lattice` package is provided for `qtpaint` by the `mosaiq` package. The `cranvas` package builds on `qtpaint` to provide a collection of high-level interactive plots in the conceptual vein of `GGobi`. A number of general utilities are implemented by `qtutils`, including an object browser widget, an R console widget, and a conventional R graphics device based on `QGraphicsView`.

While `qtbase` 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 other 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^[2], 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 `qtbase` package is available from CRAN. The package depends on the Qt framework, available as a binary install from <http://qt.nokia.com/>. Before `qtbase` is loaded, an automated attempt to install the framework is made, if it is not already present.

12.2 An introductory example

As a synopsis for how to program a GUI using `qtbase`, we present a simple dialog that allows the user to input a date. A detailed introduction to these concepts will follow this example.

The package may be loaded like any other R package:

```
library(qtbase)
```

Constructors As with all other toolkits, Qt widgets are objects, and the objects are created with constructors. 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.

¹There is a GTK+ WebKit port, but it is not included with GTK+ itself.

[2] Nokia Corporation. <http://http://doc.qt.nokia.com/>.

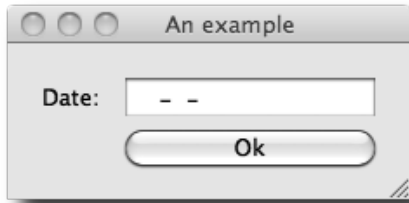


Figure 12.1: Screenshot of our sample GUI to collect a date from the user.

```

window <- Qt$QWidget()
label <- Qt$QLabel("Date:")
edit <- Qt$QLineEdit()
button <- Qt$QPushButton("Ok")

```

The constructors are found not in the global environment, but rather in the Qt environment, an object exported from the `qtbase` namespace. As such, the `$` lookup operator is used.

Widgets in Qt have various properties that specify the state of the object. For example, the `windowTitle` property controls the title of a top-level widget:

```

window$windowTitle <- "An example"

```

Qt objects are represented as extended R environments, and every property is a member of the environment. The `$` function called above is simply that for environments.

Method calls tell an object to perform some behavior. Like properties, methods are accessible from the instance environment. For example, the `QLineEdit` widget supports an input mask that constrains user input to a particular syntax. 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 help page for `QLineEdit`. To set an input mask we have:

```

edit$setInputMask("0000-00-00")

```

Layout managers Qt uses separate layout manager objects to organize widgets. This is similar to Java/Swing and `tcltk`, but not `RGtk2`. Layout managers will be discussed more thoroughly in Chapter 13, 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:

```

layout <- Qt$QGridLayout()

```

```
layout$addWidget(label, row = 0, column = 0,
                  rowspan = 1, columnSpan = 1)
layout$addWidget(edit, 0, 1, 1, 1)
layout$addWidget(button, 1, 1, 1, 1)
```

We need to attach our layout to the widget window:

```
window$setLayout(layout)
```

Finally, to view our GUI (Figure 12.1), we must call its `show` method.

```
window$show()
```

Callbacks As with other GUI toolkits, interactivity is implemented by callbacks connected to particular signals. To react to the clicking of the button, the programmer attaches a handler to the `clicked` signal using the `qconnect` function. The function requires the object, the signal name, and the handler. Here, we print the value stored in the “Date” field.

```
handler <- function() print(edit$text)
qconnect(button, "clicked", handler)
```

We will discuss callbacks more completely in Section 12.6.

Object-oriented support `QLineEdit` can validate text input, and we would like to validate the entered date. There are a few built-in validators. For this purpose the regular expression validator could be used, but it would be difficult to write a sufficiently robust expression. Instead we attempt to coerce the string value to a date via R’s `as.Date` function with a format of `“%Y-%m-%d”`. In GTK+, validation would be implemented by a signal handler or other callback. However, as C++ is object-oriented, Qt expects the programmer to derive a new class from `QValidator` and pass an instance to the `setValidator` method on `QLineEdit`.

It is possible to define R subclasses of C++ classes with `qtbases`. More details on working with classes and methods are provided in Section 12.8. For this task, we need to extend `QValidator` and override its `validate` virtual method. The `qsetClass` function defines a new class:

```
qsetClass("DateValidator", Qt$QValidator,
          function(parent = NULL) {
            super(parent)
          })
```

To override `validate`, we call `qsetMethod`:

```
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)
```

```

    return(Qt$QValidator$Invalid)
  else
    return(Qt$QValidator$Acceptable)
  })

```

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 indicates a state of “Acceptable,” “Invalid,” or, if neither can be determined, “Intermediate.” These values are listed in an enumeration in the `Qt$QValidator` class (cf. Section 12.7 for more on enumerations).

The class object, which doubles as the constructor, is defined in the current top-level environment as a side effect of `qsetMethod`. We call it to construct an instance, which is passed to the edit widget:

```

validator <- DateValidator()
edit$setValidator(validator)

```

12.3 Classes and objects

The `qtbase` 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 315 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`:

```
head(names(Qt$QWidget), n = 3)
```

```
[1] "connect"      "DrawChildren"  "DrawWindowBackground"
```

then access one of the enumeration values:

```
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` with:

² The `Qt` object is an instance of `RQtLibrary`. The `qtbase` 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.


```
widget <- Qt$QWidget()
```

The widget object has a class structure that reflects the class inheritance structure of Qt:

```
class(widget)
```

```
[1] "QWidget"          "QObject"          "QPaintDevice"  
[4] "UserDefinedDatabase" "environment"      "RQtObject"
```

The base class, RQtObject, is an environment containing the properties and methods of the instance. For widget, we list the first few using ls:

```
head(ls(widget), n=3)
```

```
[1] "mapFromParent" "setContextMenuPolicy" "showMinimized"
```

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 R's completion mechanism works (?rcompgen).) For example, a QWidget has a windowTitle property which is used when the widget draws itself with a window:

```
widget$windowTitle # initially NULL
```

```
NULL
```

```
widget$windowTitle <- "a new title" # set property  
widget$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:

```
widget$show()
```

The environment structure of the object masks the fact that the properties and methods can be defined in a parent class of the object. For example, a button widget is provided by the QPushButton constructor, as in

```
button <- Qt$QPushButton()
```

QPushButton extends QWidget and thus inherits the properties like visible:

```
is(button, "QWidget")
```

```
[1] TRUE
```

```
button$visible
```

```
[1] FALSE
```

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.

12.4 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:

```
button <- 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`:

```
widget <- Qt$QWidget()
button <- Qt$QPushButton(widget)
```

An alternative constructor for `QPushButton` accepts the text for the label on the button:

```
button <- Qt$QPushButton("Button text")
```

Buttons can also have icons. for example,

```
style <- Qt$QApplication$style()
icon <- style$standardIcon(Qt$QStyle$SP_DialogOkButton)
button <- Qt$QPushButton(icon, "Ok")
```

We have passed three types of objects as the first argument to `QPushButton`: a `QWidget`, a string, and finally a `QIcon`. The dispatch depends only on the type of argument, unlike the constructors in `RGtk2`, which dispatches based on which arguments are specified. (In particular, dispatch in Qt is based on position of argument, but not on names given to arguments. We use names only for clarity in our examples.)

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, to learn the methods for a simple button, we would call:

```
method_info <- qmethods(Qt$QPushButton)
dim(method_info)
```

```
[1] 431 6
```

```
head(method_info[,1:3], n = 3)
```

	name	return	signature
1	QPushButton	QPushButton*	QPushButton()
2	QPushButton	QPushButton*	QPushButton(QWidget*)
3	QPushButton	QPushButton*	QPushButton(QIcon, QString)

12.5 Properties

Every QObject, which includes every widget, may declare a set of properties that represents its state. We list some of the available properties for our button:

```
head(qproperties(button))
```

	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 writable. Virtually every property value can 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:

```
button$objectName <- "My button"
button$objectName
```

```
[1] "My button"
```

```
button$modal
```

```
[1] FALSE
```

```
cat(try(button$modal <- TRUE))
```

```
Error in button$modal <- TRUE : Property 'modal' is read-only
```

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:

```
button$setObjectName("My button")
```

However, it is recommended to use the replacement syntax shown in the previous example, for the sake of symmetry.

12.6 Signals

Qt 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 us 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:

```
button <- Qt$QPushButton("click me")
qconnect(button, "clicked", function() message("ouch"))
button$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), n = 5)
```

	name	signature
4	pressed	pressed()
5	released	released()
6	clicked	clicked(bool)
7	clicked	clicked()
8	toggled	toggled(bool)

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. Rarely, two signatures will have the same number of arguments, in which case 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, there are two signatures for the `clicked` signal: `clicked()` and `clicked(bool)`. Even if we specify only `clicked` as the signal name, the `clicked(bool)` signature is chosen, since our handler has a single argument. Thus, these two calls are equivalent:

```
qconnect(button, "clicked", function(checked) print(checked))
qconnect(button, "clicked(bool)",
         function(checked) print(checked))
```

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, although we encourage the use of closures for this purpose.

Disconnecting or blocking signals 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(button, "clicked",
                 function() message("ouch"))
button$disconnect(proxy)
```

```
[1] TRUE
```

The above will permanently disconnect the signal handler. To block all of the signals emitted by a particular `QObject` temporarily, call the `blockSignals` method. The method takes a logical value indicating whether the signals should be blocked.

Hardware events Unlike `GTK+`, `Qt` widgets generally do not emit hardware events, such as a mouse-press event, via signals. Instead, a method in the widget is invoked upon receipt of an event. The developer is expected to extend the widget's class 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 12.8.

12.7 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
```

```
Enum value: Expanding (7)
```

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.

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:

```
label <- Qt$QLabel("Our text")
label$alignment <- Qt$Qt$AlignRight | Qt$Qt$AlignTop
```

To check if the alignment is to the right, we could query by:

```
as.logical(label$alignment & Qt$Qt$AlignRight)
```

```
[1] TRUE
```

12.8 Extending Qt classes from R

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 necessary to extend classes only when we need to alter the behavior of a widget fundamentally (cf. Chapter 11). The `qtbase` package allows the R user to extend C++ classes in order to enhance the features of Qt. The `qtbase` package includes functions `qsetClass` and `qsetMethod` to create subclasses and their methods. Methods may override virtual methods in an ancestor C++ class, and C++ code will invoke the R implementation when calling the overridden virtual. A property may be

defined with a getter and setter function. If a type is specified, and the class derives from `QObject`, the property will be exposed by Qt. It is also possible to store arbitrary objects in an instance of an R class; we will refer to these as dynamic fields. They are private to the class but are otherwise similar to attributes on any R object. Their type is not checked, and they are useful as storage mechanisms for implementing properties.

Defining a class

Here, we show a generic example and follow with a specific one.

```
qsetClass("SubClass", Qt$QWidget)
```

This creates a variable named `SubClass` in the workspace:

```
SubClass
```

```
Class 'R::.GlobalEnv::SubClass' with 315 public methods
```

Its value is an `RQtClass` object that behaves like the `RQtClass` for the built-in classes, such as `Qt$QWidget`. There are no static methods or enumerations in an R class, so the class object is essentially the constructor:

```
instance <- SubClass()
```

By default, the constructor delegates directly to the constructor in the parent class. A custom constructor is often useful, for example, to initialize fields or to make a compound widget. The function implementing the constructor should be passed as the `constructor` argument. By convention, `QObject` subclasses should provide a parent constructor argument for specifying the parent object. A typical usage would be:

```
qsetClass("SubClass2", Qt$QWidget ,
         function(property, parent = NULL) {
           super(parent)
           this$property <- property
         })
```

Within the body of a constructor, the `super` variable refers to the constructor of the parent class, often called the “super” class. In the above, we call `super` to delegate the registration of the parent to the `QWidget` constructor. Another special symbol in the body of a constructor is `this`, which refers to the instance being constructed. We can set and implicitly create fields in the instance by using the same syntax as setting properties.

Defining methods

We can define new methods, or override methods from a base class, through the `qsetMethod` function. For example, accessors for a field can be defined with:

```
qsetMethod("field", SubClass, function() field)
qsetMethod("setField", SubClass, function(value) {
  this$field <- value
})
```

For an override of an existing method to be visible from C++, the method must be declared virtual in C++. The access argument specifies the scope of the method: "public" (default), "protected", or "private". These have the same meaning as in C++.

As with a constructor, the symbol `this` in a method definition refers to the instance. There is also a `super` function that behaves similarly to the `super` found in a constructor: it searches for an inherited method of a given name and invokes it with the passed arguments:

```
qsetMethod("setVisible", SubClass, function(value) {
  message("Visible: ", value)
  super("setVisible", value)
})
```

In the above, we intercept the setting of the visibility of our widget. If we hide or show the widget, we will receive a notification to the console:

```
instance$show()
```

This is somewhat similar to the behavior of `callNextMethod`, except `super` is not restricted to calling the same method.

Defining signals and slots

Two special types of methods are slots and signals, introduced earlier in the chapter. These exist only for `QObject` derivatives. Most useful are signals. Here we define a signal:

```
qsetSignal("somethingHappened", SubClass)
```

If the signal takes an argument, we need to indicate that in the signature:

```
qsetSignal("somethingHappenedAtIndex(int)", SubClass)
```

Writing a signature requires some familiarity with C/C++ types and syntax, but this is concise and consistent with how Qt describes its methods. Although almost always public, it is possible to make a signal protected or private, via the access argument.

Defining a slot is very similar to defining a signal, except a method implementation must be provided as an R function:

```
qsetSlot("doSomethingToIndex(int)", SubClass, function(index) {
  # ....
})
```


The advantage of a slot compared to a method is that a slot is exposed to the Qt metaobject system. This means that a slot could be called from another dynamic environment, like from Javascript running in the QScript module or via the D-Bus through the QDBus module. It is also necessary to use slots as signal handlers for a GUI built with QtDesigner, if we are using the automated connection feature (see Section 12.10).

Defining properties

A property, introduced earlier, is a self-describing field that is encapsulated by a getter and a setter. We can define a property on any class using the `qsetProperty` function. Here is the simplest usage:

```
qsetProperty("property", SubClass)
```

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

We can now access property like any other property; for example:

```
instance <- SubClass()
instance$property # initially NULL
```

```
NULL
```

```
instance$property <- "value"
instance$property
```

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

However, the property is not actually exposed by Qt to dynamic systems, like the QtScript Javascript engine, which would understand only Qt types. To export a property, we must provide the type argument, which is covered later.

By default, the property value is actually stored as a (private) field in the object, named by the format `".property"`. We can override the default behavior by passing a getter and/or setter function to the `read` and/or `write` arguments, respectively. For example:

```
qsetProperty("checkedProperty", SubClass, write=function(x) {
  if (!is(x, "character"))
    stop("'checkedProperty' must be a character vector")
  this$.checkedProperty <- x
})
```

We have taken advantage of the setter override to check the validity of the incoming value. If `NULL` is passed as the `write` argument, the property is read-only. We might also want to override the `read` function, for cases where a property depends only on other properties or on some external resource.

To emit a signal automatically whenever a property is set, we can pass the name of the signal to the `notify` argument of `qsetProperty`:

```
qsetSignal("propertyChanged", SubClass)
qsetProperty("property", SubClass, notify = "propertyChanged")
```

If a class derives from `QObject`, as does any widget, we can specify the C++ type of the property to expose it to the Qt meta object system:

```
qsetProperty("typedProperty", SubClass, type = "QString")
```

```
tail(qproperties(SubClass()), 1)
```

		type	readable	writable
typedProperty	QString	\\\\x84\\x9c\\020	TRUE	TRUE

The type is now exposed via the general `qproperties` function. Specifying the type enables all of the features of a Qt property.

Example 12.1: A watcher for workspace objects

Qt provides the `QFileSystemWatcher` class for monitoring changes to the underlying file system. Here, we create an analogous component that monitors changes to the global workspace. With `gWidgets` (cf. Example 4.9), we implemented the observer pattern to notify listeners for changes to the workspace. With Qt, we can leverage the existing signal framework. This example demonstrates only the watcher; implementing a view is left to Example 15.1.

Our basic model subclasses `QObject`, not `QWidget`, as it has no graphical representation – a job left for its views:

```
qsetClass("WSWatcher", Qt$QObject, function(parent = NULL) {
  super(parent)
  updateVariables()
})
```

We have two main properties: a list of workspace objects and a digest hash for each, which we use for comparison purposes. The digest is generated by the `digest` package, which we load:

```
library(digest)
```

We store the digests in a property:

```
qsetProperty("digests", WSWatcher)
```

When a new object is added, an object is deleted, or an object is changed, we wish to signal that occurrence to any views of the model. For that purpose, we define a new signal below:

```
qsetSignal("objectsChanged", WSWatcher)
```

We then pass this signal name to the `notify` argument when defining the `objects` property, so that assignment will emit the signal:

```
qsetProperty("objects", WSWatcher, notify = "objectsChanged")
```

To monitor changes, we keep track of the digest values and names of the old objects:

```
qsetProperty("old_digests", WSWatcher)
qsetProperty("old_objects", WSWatcher)
```

Our class has a few methods defined for it. We need one to update the variable list. This implementation simply compares the digest of the current workspace objects with a cached list. If there are differences, we update the objects, which will in turn signal a change.

```
qsetMethod("updateVariables", WSWatcher, function() {
  x <- sort(ls(envir = .GlobalEnv))
  objs <- sapply(mget(x, .GlobalEnv), digest)

  if((length(objs) != length(digests)) ||
      length(digests) == 0 ||
      any(objs != digests)) {
    this$old_digests <- digests           # old
    this$old_objects <- objects
    this$digests <- objs                 # update cache
    this$objects <- x                   # emits signal
  }
  invisible()
})
```

For convenience to any user of this class, we define two more methods: one to indicate which objects were changed and one to indicate which objects were added:

```
qsetMethod("changedVariables", WSWatcher, function() {
  changed <- setdiff(old_digests, digests)
  old_objects[old_digests %in% changed]
})
##
qsetMethod("addedVariables", WSWatcher, function() {
  added <- setdiff(digests, old_digests)
  objects[digests %in% added]
})
```

Finally, we arrange to call our update function as needed. If the workspace size is modest, using a task callback is a reasonable strategy:

```
watcher <- WSWatcher() # an instance
addTaskCallback(function(expr, value, ok, visible) {
  watcher$updateVariables()
```

```
TRUE
})
```

Another alternative would be to use a timer to call the `updateVariables` method periodically:

```
timer <- Qt$QTimer()
timer$setSingleShot(FALSE)           # or TRUE for run once
qconnect(timer, "timeout", function() watcher$updateVariables())
timer$start(as.integer(3*1000))      # 3 seconds
```

To illustrate, we connect a handler to the `objectsChanged` signal and expect the handler to be invoked when we create a `new_object` in the workspace:

```
qconnect(watcher, "objectsChanged", function()
  message("workspace objects were updated"))
new_object <- "The change should be announced"
```

```
workspace objects were updated
```

12.9 QWidget basics

The widgets we discuss in the next section 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 hierarchy. It implements the event processing and property systems. The `QWidget` class is the base class for all widgets and implements their shared functionality.

Upon construction, widgets are invisible, so that they may be configured behind the scenes. The `visible` property controls whether a widget is visible.

```
widget <- Qt$QWidget()
widget$visible
```

```
[1] FALSE
```

The `show` and `hide` methods are the corresponding convenience functions for making a widget visible and invisible, respectively.

```
widget$show()
```

```
widget$visible
```

```
[1] TRUE
```

```
widget$hide()
```

```
widget$visible
```

```
[1] FALSE
```

There is an `S3` method for `print` on `QWidget` that invokes `show`. Whenever a widget is shown, all of its children are also made visible. The method `raise` will raise the window to the top of the stack of windows.

Similarly, the property `enabled` controls whether a widget is sensitive to user input, including mouse events.

```
button <- Qt$QPushButton("button")
button$enabled <- FALSE
```

Only one widget can have the keyboard focus at once. The user shifts the focus by `tab-key` navigation or mouse clicks (although this behavior can be customized, cf. `focusPolicy`). When a widget has the focus, its `focus` property is `TRUE`. The property is read-only; the focus is shifted programmatically to a widget by calling its `setFocus` method.

Qt has a number of mechanisms for the user to query a widget for some description of its purpose and usage. Tooltips, stored as a string in the `toolTip` property, may be shown when the mouse hovers over the widget. Similarly, the `statusTip` property holds a string to be shown in the status bar instead of in a pop-up window. Finally, Qt provides a “What’s This?” tool that will show the text in the `whatsThis` property in response to a query, such as pressing `SHIFT+F1` when the widget has focus.

Except for top-level windows, the position and size of a widget are determined automatically by a layout algorithm; see Chapter 13. To specify the size of a top-level window, manipulate the `size` property, which holds a `QSize` object:

```
widget$size <- qsize(400, 400)
## or
widget$resize(400, 400)
widget$show()
```

We create the `QSize` object with the `qsize` convenience function implemented by the `qtbase` package. The `resize` method is another convenient shortcut. We should generally configure the size of a window before showing it, as this helps the window manager place the window optimally.

Fonts

Fonts in Qt are represented by the `QFont` class. The `qtbase` package defines a convenience constructor for `QFont` called `qfont`. The constructor accepts 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 font should be italicized, as a logical. Defaults are obtained from the application font, returned by `Qt$QApplication$font()`.

For example, we could create a 12-point, bold, italicized font from the Helvetica family with:

```
font <- QFont(family = "helvetica", fontsize = 12,
             weight = Qt$QFont$Bold, italic = TRUE)
```

The font for a widget is stored in the font property. For example, we change the font for a label:

```
label <- Qt$QLabel("Text for the label")
label$font <- font
```

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.

To discover which fonts are available from the windowing system, construct a QFontDatabase and call its methods, like `families`, `pointSizes`, `styles`, etc.

Styles

Palette Every platform has its own distinct look and feel, and an application should generally conform to platform conventions. Qt hides these details from the application. Every widget has a palette, stored in its `palette` property and represented by a QPalette object. A QPalette maps the state of a widget to a group of colors that is used for painting the widget. The possible states are `active`, `inactive`, and `disabled`. Each color within a group has a specific role, as enumerated in `QPalette::ColorRole`. Examples include the color for the background (`Window`), the foreground (`WindowText`), and the selected state (`Highlight`). Qt chooses the correct default palette depending on the platform and the type of widget. We can change the colors used in rendering a widget by manipulating the palette.

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. In Qt it is also possible to customize the rendering of a widget using CSS syntax. The supported syntax is described in the overview on style sheets provided with Qt documentation and is not summarized here, as it is quite readable.

The style sheet for a widget is stored in its `styleSheet` property, as a string. For example, for a button, we could set the background to white and the foreground to red (see Figure 12.2):

```
button <- Qt$QPushButton("Style sheet example")
button$show()
button$styleSheet <-
  "QPushButton {color: red; background: white}"
```

The CSS syntax may be unfamiliar to R programmers, so the `qtbases` package provides an alternative interface that is reminiscent of the `par` function. We specify the above style sheet in this syntax:

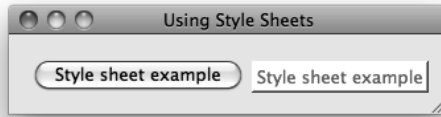


Figure 12.2: Styling a widget with a style sheet can dramatically alter its appearance.

```
q.setStyleSheet(color = "red", background = "white",  
               what = "QPushButton", widget = button)
```

The widget argument defaults to `NULL`, which applies the style sheet to every widget matching `what` in the application. The default for `what` is `"*"`, meaning that the style sheet applies to any widget class. The following would cause all widgets in the application to have the same colors as the button:

```
q.setStyleSheet(color = "red", background = "white")
```

Example 12.2: An 'error label'

This example extends the line-edit widget to display an error state via an icon embedded within the entry box. Such a widget might prove useful when we are validating entered values. Our implementation uses a style sheet to place the icon in the background and to prevent the text from overlapping the icon.

To indicate an error, we will add an icon and set the tooltip to display an informative message (Figure 12.3). The constructor will be the default, so our class is defined with:

```
qsetClass("LineEditWithError", Qt$QLineEdit)
```

The main method sets the error state. We use style sheets to place an image to the left of the entry message and set the tooltip.

```
qsetMethod("setError", LineEditWithError, function(msg) {  
  file <- system.file("images/cancel.gif", package="gWidgets")  
  q.setStyleSheet("background-image" = sprintf("url(%s)", file),  
                 "background-repeat" = "no-repeat",  
                 "background-position" = "left top",  
                 "padding-left" = "20px",  
                 widget = this)  
  setToolTip(msg)  
})
```

We can clear the error by resetting the properties to `NULL`.

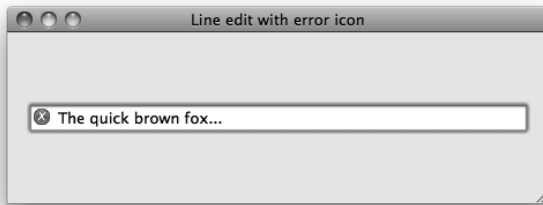


Figure 12.3: Using a style sheet to customize the line-edit class to show an error indicator.

```
qsetMethod("clearError", LineEditWithError, function() {
  setStyleSheet(NULL)
  setToolTip(NULL)
})
```

```
edit <- LineEditWithError()
edit$text <- "The quick brown fox..."
edit$setError("Replace with better boilerplate text")
edit$clearError()
```

12.10 Importing a GUI from QtDesigner

QtDesigner is a tool for graphical, drag-and-drop design of GUI forms. Although this book focuses on constructing a GUI by programming in R, we recognize that a graphical approach may be preferable in some circumstances. QtDesigner outputs a GUI definition as an XML file in the "UI" format. The `QUiLoader` class loads a "UI" definition³ through its `load` method:

```
loader <- Qt$QUiLoader()
widget <- loader$load(Qt$QFile("textfinder.ui"))
```

The widget object could be shown directly; however, we first need to implement the behavior of the GUI by connecting to signals. Through the QtDesigner GUI, the user can connect signals to slots on built-in widgets. This works for some trivial cases, but in general we need to handle signals with R code. There are two ways to accomplish this: manual and automatic.

To connect an R handler to a signal manually, we first need to obtain the widget with the signal. Every widget in a UI file is named, so we can

³The `textfinder.ui` file was taken from the Qt Text Finder example at <http://doc.qt.nokia.com/4.7-snapshot/uitools-textfinder.html>.

call the `qfindChild` utility function to find a specific widget. Assume we have a button named “findButton” and corresponding text entry “lineEdit” in our UI file, then we retrieve them with:

```
find_button <- qfindChild(widget, "findButton") # by name
line_edit <- qfindChild(widget, "lineEdit")
```

Then we connect to the clicked signal:

```
qconnect(find_button, "clicked", function() {
  findText(line_edit$text)
})
```

Alternatively, we could establish the signal connections automatically. This requires defining each signal handler to be a slot in the parent object, which will need to be of a custom class:

```
qsetClass("MyMainWindow", Qt$QWidget, function() {
  loader <- Qt$QUiLoader()
  widget <- loader$load(Qt$QFile("textfinder.ui"), this)
  Qt$QMetaObject$connectSlotsByName(this)
})
```

The constructor first loads the UI definition, with the main window as the parent for the loaded interface. It then calls `connectSlotsByName` to establish the connections automatically. This descends the widget hierarchy, attempting to match signals in the descendants to slots in the top-level widget. For a slot to be connected to the correct signal, it must be named according to the convention `"on_[objectName]_[signalName]"`. For example,

```
qsetSlot("on_findButton_clicked", MyMainWindow, function() {
  findText(line_edit$text)
})
```

defines a handler for the `clicked` signal on `findButton`. Finally, the signal handler connection is established upon construction of the main window:

```
MyMainWindow()
```

In the case of a large, complex GUI, this automatic approach is probably more convenient than manually establishing the connections.

Qt: Layout Managers and Containers

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.

This chapter will introduce the available layout managers, of which there are three types: `box` (`QBoxLayout`), `grid` (`QGridLayout`), and `form` (`QFormLayout`). Widgets that function primarily as containers, such as the `frame` and `notebook`, are also described here.

Example 13.1: Synopsis of layouts in Qt

This example uses a combination of different layout managers to organize a reasonably complex GUI. It serves as a synopsis of the layout functionality in Qt. A more gradual and detailed introduction will follow this example. Figure 13.1 shows a screenshot of the finished layout.

First, we need a widget as the top-level container. We assign a grid layout to the window for arranging the main components of the application:

```
window <- Qt$QWidget()
window$setWindowTitle("Layout example")
grid_layout <- Qt$QGridLayout()
window$setLayout(grid_layout)
```

There are three objects managed by the grid layout: a table (we use a label as a placeholder), a notebook, and a horizontal box layout for some buttons. We construct them with:

```
fake_table <- Qt$QLabel("Table widget")
notebook <- Qt$QTabWidget()
button_layout <- Qt$QHBoxLayout()
```

Then add them to the grid layout:

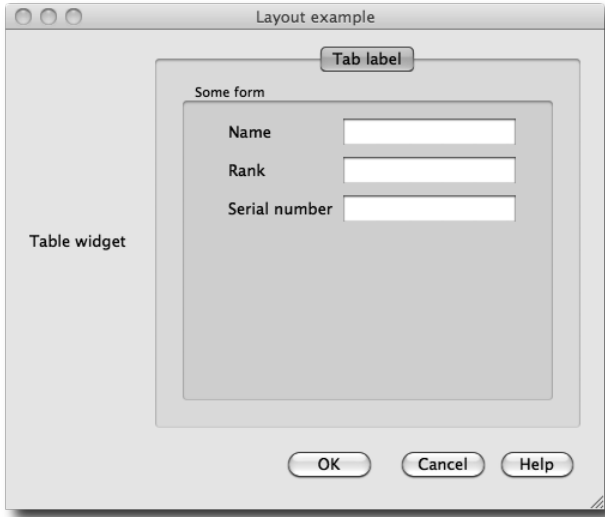


Figure 13.1: A mock GUI illustrating various layout managers provided by Qt.

```
grid_layout$addWidget(fake_table, row=0, column=0,
                      rowspan=1, colspan=1)
grid_layout$addWidget(notebook, 0, 1)
grid_layout$addLayout(button_layout, 1, 1)
```

Next, we construct our buttons and add them to the box putting 12 pixels of space between the last two.

```
b <- sapply(c("OK", "Cancel", "Help"),
           function(i) Qt::QPushButton(i))
button_layout$setDirection(Qt::QBoxLayout::RightToLeft)
button_layout$addStretch(1L) # stretch
button_layout$addWidget(b$OK)
button_layout$addWidget(b$Cancel)
button_layout$addSpacing(12L) # spacing
button_layout$addWidget(b$Help)
```

We added a stretch, which acts much like a spring, to pack our buttons against the right side of the box. A fixed space of 12 pixels is inserted between the “Cancel” and “Help” buttons.

The notebook widget is constructed next, with a single page:

```
notebook_page <- Qt::QWidget()
notebook$addTab(notebook_page, "Tab label")
notebook$setTabToolTip(0, "A notebook page with a form")
```

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. We place a form layout in the notebook page and populate it:

```
form_layout <- Qt$QFormLayout()
notebook_page$setLayout(form_layout)
l <- sapply(c("name", "rank", "snumber"), Qt$QLineEdit)
form_layout$addRow("Name", l$name)
form_layout$addRow("Rank", l$rank)
form_layout$addRow("Serial number", l$snumber)
```

Each `addRow` call adds a label and an adjacent input widget, in this case a text entry.

This concludes our cursory demonstration of layout in Qt. We have constructed a mock-up of a typical application layout using the box, grid, and form layout managers.

13.1 Layout basics

Adding and manipulating child components

We will demonstrate the basics of layout in Qt with a horizontal box layout, `QHBoxLayout`:

```
layout <- Qt$QHBoxLayout()
```

`QHBoxLayout`, like all other layouts, is derived from the `QLayout` base class. Details specific to box layouts are presented in Section 13.2.

A layout is not a widget. Instead, a layout is set on a widget, and the widget delegates the layout of its children to the layout object:

```
widget <- Qt$QWidget()
widget$setLayout(layout)
```

Child widgets are added to a container through the `addWidget` method:

```
layout$addWidget(Qt$QPushButton("Push Me"))
```

In addition to adding child widgets, we can nest child layouts by calling `addLayout`.

Internally, layouts store child components as items of class `QLayoutItem`. The item at a given index (0-based) is returned by the `itemAt` method. We get the first item in our layout:

```
item <- layout$itemAt(0)
```

The actual child widget is retrieved by calling the `widget` method on the item:

```
button <- item$widget()
```

Qt provides the methods `removeItem` and `removeWidget` to remove an item or widget from a layout:

```
layout$removeWidget(button)
```

Although the widget is no longer managed by a layout, its parent widget is unchanged. The widget will not be destroyed (removed from memory) as long as it has a parent. Thus, to destroy a widget, we should set the parent of the widget `NULL` using `setParent`:

```
button$setParent(NULL)
```

Size and space negotiation

The allocation of space to child widgets depends on several factors. The Qt documentation for layouts spells out the steps well: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 get more space only 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 size or minimum size hint, in which case the stretch factor is their determining factor.)
5. 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`.

¹<http://doc.qt.nokia.com/4.7/layout.html>

Table 13.1: Possible size policies from `QSizePolicy`

Policy	Meaning
Fixed	Require the size hint exactly
Minimum	Treat the size hint as the minimum, allowing expansion
Maximum	Treat the size hint as the maximum, allowing shrinkage
Preferred	Request the size hint, but allow for either expansion or shrinkage
Expanding	Treat like Preferred, except the widget desires as much space as possible
MinimumExpanding	Treat like Minimum, except the widget desires as much space as possible
Ignored	Ignore the size hint and request as much space as possible

It contains a separate policy value, taken from the `QSizePolicy` enumeration, for the vertical and horizontal directions. If a layout is set on a widget, then the widget inherits its size policy from the layout. The possible size policies are listed in Table 13.1.

As an example, consider `QPushButton`. It is the convention that a button will allow horizontal, but not vertical, expansion. It also requires 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:

```
button <- Qt::QPushButton("No expansion")
button$setSizePolicy(vertical = Qt::QSizePolicy$Fixed,
                    horizontal = Qt::QSizePolicy$Fixed)
```

Thus, the sizing behavior is largely inherent to the widget or its layout, if any, rather than any parent layout parameters. This is a major difference from `GTK+`, where a widget can request only 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.

Most widgets attempt to fill the allocated space; however, this is not always appropriate, as in the case of labels. In such cases, the widget will not expand and needs to be aligned within its space. By default, the widget is centered. We can control the alignment of a widget via the `setAlignment` method. For example, we align the label to the left side of the layout through:

```
label <- Qt$QLabel("Label")
layout$addWidget(label)
layout$setAlignment(label, Qt$Qt$AlignLeft)
```

Alignment is also possible to the top, bottom, and right. The alignment values are flags and may be combined with | to specify both vertical and horizontal alignment.

The spacing between every cell of the layout is in the spacing property. The following requests five pixels of space:

```
layout$spacing <- 5L
```

13.2 Box layouts

Box layouts arrange child widgets as if they were packed into a box in either the horizontal or vertical orientation. The `QHBoxLayout` class implements a horizontal layout, whereas `QVBoxLayout` provides a vertical one. Both of these classes extend the `QBoxLayout` class, where most of the functionality is documented. We create a horizontal layout and place it in a window:

```
hbox <- Qt$QHBoxLayout()
widget <- Qt$QWidget()
w$setLayout(hbox)
```

Child widgets are added to a box container through the `addWidget` method:

```
buttons <- sapply(letters[1:3], Qt$QPushButton)
sapply(buttons, hbox$addWidget)
```

The `direction` property specifies the direction in which the widgets are added to the layout. By default, this is left to right (top to bottom for a vertical box).

The `addWidget` method for a box layout takes two optional parameters: the stretch factor and the alignment. Stretch factors proportionally allocate space to widgets when they expand.² However, recall that the widget size policy and hint can alter the effect of a stretch factor. After the child has been added, the stretch factor can be modified with `setStretchFactor`:

```
hbox$setStretchFactor(buttons[[1]], 2.0)
```

If the layout later grows horizontally, the first button will grow (stretch) at twice the rate of the other buttons.

²For those familiar with GTK+, the difference between a stretch factor of 0 and 1 is roughly equivalent to the difference between "FALSE" and "TRUE" for the value of the `expand` parameter to `gtkBoxPackStart`.

Spacing There are two types of spacing between two children: fixed and expanding. Fixed spacing between any two children was already described. To add a fixed amount of space between two specific children, call the `addSpacing` method while populating the container. The following line is from Example 13.1:

```
hbox$addSpacing(12L)
hbox$addWidget(Qt$QPushButton("d"))
```

We have placed a gap of twelve pixels between button "c" and the new button "d".

An expanding, spring-like spacer between two widgets is known as a *stretch*. We add a stretch with a factor of 2.0 and subsequently add a child button that will be pressed against the right side of the box as the layout grows horizontally:

```
hbox$addStretch(2)
hbox$addWidget(Qt$QPushButton("Help..."))
```

This is just a convenience for adding an invisible widget with some stretch factor.

Struts It is sometimes desirable to restrict the minimum size of a layout in the perpendicular direction. For a horizontal box, this is the height. The box layout provides the *strut* for this purpose:

```
hbox$addStrut(10) # at least 10 pixels high
```

13.3 Grid layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns. To illustrate grid layouts we mock up a GUI centered around a text area widget (Figure 13.2). To begin, we create the window with the grid layout:

```
window <- Qt$QWidget()
window$setWindowTitle("Layout example")
layout <- Qt$QGridLayout()
window$setLayout(layout)
```

When we add a child to the grid layout, we need to specify the zero-based row and column indices:

```
layout$addWidget(Qt$QLabel("Entry:"), 0, 0)
layout$addWidget(Qt$QLineEdit(), 0, 1, rowspan = 1, colspan=2)
```

In the second call to `addWidget`, we pass values to the optional arguments for the row and column span. These are the numbers of rows and columns, respectively, that are spanned by the child. For our second row, we add a labeled combo box:

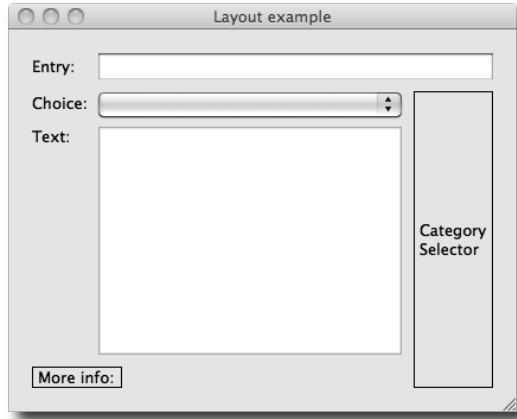


Figure 13.2: A mocked up layout using the `QGridLayout` class. There are three columns and four non-homogeneous rows; in addition, several child components span more than one cell.

```
layout$addWidget(Qt$QLabel("Choice:"), 1, 0)
layout$addWidget(Qt$QComboBox(), 1, 1)
```

The bottom three cells in the third column are managed by a sub-layout, in this case a vertical box layout. We use a label as a stub and set a frame style to have it stand out:

```
layout$addLayout(sub_layout <- Qt$QVBoxLayout(),
                 1, 2, rowspan=3, 1)
sub_layout$addWidget(label <- Qt$QLabel("Category\nSelector"))
label$setFrameStyle(Qt$QFrame$Box)
```

The text-edit widget is added to the third row, second column:

```
layout$addWidget(Qt$QLabel("Text:"), 2, 0, Qt$Qt$AlignTop)
layout$addWidget(edit <- Qt$QTextEdit(), 2, 1)
```

Since this widget will expand, we align the label to the top of its cell. Otherwise, it will be centered in the vertical direction.

Finally we add a space for information on the fourth row:

```
layout$addWidget(label <- Qt$QLabel("More info:"), 3, 0,
                 rowspan = 1, colspan = 2)
label$setSizePolicy(Qt$QSizePolicy$Fixed,
                   Qt$QSizePolicy$Preferred)
label$setFrameStyle(Qt$QFrame$Box)
```

Again we draw a frame around the label. By default the box would expand to fill the space of the two columns, but we prevent this through a "Fixed" size policy.

There are a number of parameters controlling the sizing and spacing of the rows and columns. The concepts apply equivalently to both rows and columns, so we will limit our discussion to columns, without loss of generality. A minimum width is set through `setColumnMinimumWidth`. The actual minimum width will be increased, if necessary, to satisfy the minimal width requirements of the widgets in the column. If more space is available to a column than requested, the extra space is apportioned according to the stretch factors. A column stretch factor is set by calling the `setColumnStretch` method.

Since there are no stretch factors set in our example, the space allocated to each row and column would be identical when resized. To allocate extra space to the text area, we set a positive stretch factor for the third row and second column:

```
layout$setRowStretch(2, 1)           # third row
layout$setColumnStretch(1,1)        # second column
```

As it is the only item with a positive stretch factor, it will be the only widget to expand when the parent widget is resized.

The spacing between widgets can be set in both directions via the `spacing` property, or set for a particular direction with `setHorizontalSpacing` or `setVerticalSpacing`. The default values are derived from the style.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column:

```
edit <- layout$itemAtPosition(0, 1)$widget()
```

The `item` method `widget` returns the corresponding widget. Removing a widget is similar to a box layout, using `removeItem`, or `removeWidget`. The methods `rowCount` and `columnCount` return the dimensions of the grid.

13.4 Form layouts

Forms can easily be arranged with the grid layout, but Qt provides a convenient high-level form layout (`QFormLayout`) that conforms to platform-specific conventions. A form consists of a number of rows, where each row has a label and an input widget. We create a form and add some rows for gathering parameters to the `dnorm` function:

```
window <- Qt$QWidget()
window$setWindowTitle("Wrapper for 'dnorm' function")
window$setLayout(layout <- Qt$QFormLayout())
sapply(c("quantile", "mean", "sd"), function(Statistic) {
  layout$addRow(Statistic, Qt$QLineEdit())
})
layout$addRow(Qt$QCheckBox("log"))
```

The first three calls to `addRow` take a string for the label and a text entry for entering a numeric value. Any widget could serve as the label. A field may be any widget or layout. The final call to `addRow` places only a single widget in the row. As with other layouts, we could call `setSpacing` to adjust the spacing between rows.

To retrieve a widget from the layout, call the `itemAt` method, passing the 0-based row index and the role of the desired widget. Here, we obtain the edit box for the quantile parameter:

```
item <- layout$itemAt(0, Qt$QFormLayout$FieldRole)
quantile_edit <- item$widget()
```

13.5 Frames

The frame widget, `QGroupBox`, groups conceptually related widgets by drawing a border around them and displaying a title. `QGroupBox` is often used to group radio buttons (see Section 14.5 for an example). The title, stored in the `title` property, may be aligned to left, right or center, depending on the `alignment` property. If the `checkable` property is `TRUE`, the contents can have their sensitivity to events toggled by clicking an accompanying check button.

13.6 Separators

Like frames, a horizontal or vertical line is also useful for visually separating widgets into conceptual groups. There is no explicit line or separator widget in Qt. Rather, we configure the more general widget `QFrame`, which draws a frame around its children. Somewhat against intuition, a frame can take the shape of a line:

```
separator <- Qt$QFrame()
separator$frameShape <- Qt$QFrame$HLine
```

This yields a horizontal separator. A frame shape of `Qt$QFrame$VLine` would produce a vertical separator.

13.7 Notebooks

A notebook container is provided by the class `QTabWidget`:

```
notebook <- Qt$QTabWidget()
```

To create a page, we need to specify the label for the tab and the widget to display when the page is active:

```

notebook$addTab(Qt$QPushButton("page 1"), "page 1")
icon <- Qt$QIcon("small-R-logo.jpg")
notebook$addTab(Qt$QPushButton("page 2"), icon, "page 2")

```

As shown in the second call to `addTab`, we can provide an icon to display next to the tab label. We can also add a tooltip for a specific tab, using 0-based indexing:

```

notebook$setTabToolTip(0, "This is the first page")

```

The `currentIndex` property holds the 0-based index of the active tab. We make the second tab active:

```

notebook$currentIndex <- 1

```

The tabs can be positioned on any of the four sides of the notebook; this depends on the `tabPosition` property. By default, the tabs are on top, or "North". We move them to the bottom:

```

notebook$tabPosition <- Qt$QTabWidget$South

```

Other features include close buttons, movable pages by drag-and-drop, and scroll buttons for when the number of tabs exceeds the available space. We enable all of these:

```

notebook$tabsClosable <- TRUE
qconnect(notebook, "tabCloseRequested", function(index) {
  notebook$removeTab(index)
})
notebook$movable <- TRUE
notebook$usesScrollButtons <- TRUE

```

We would need to connect to the `tabCloseRequested` signal to actually close the tab when the close button is clicked.

Example 13.2: A help-page browser

This example shows how to create a help browser using the `QWebView` class to show web pages. The only method from this class we use is `setUrl`. The key to this is informing `browseURL` to open web pages using an R function, as opposed to the default system browser.

```

qsetClass("HelpBrowser", Qt$QTabWidget, function(parent=NULL){
  super(parent)
  #
  this$tabsClosable <- TRUE
  qconnect(this, "tabCloseRequested", function(index) {
    this$removeTab(index)
  })
  this$movable <- TRUE; this$usesScrollButtons <- TRUE
  #
  this$browser <- getOption("browser")

```

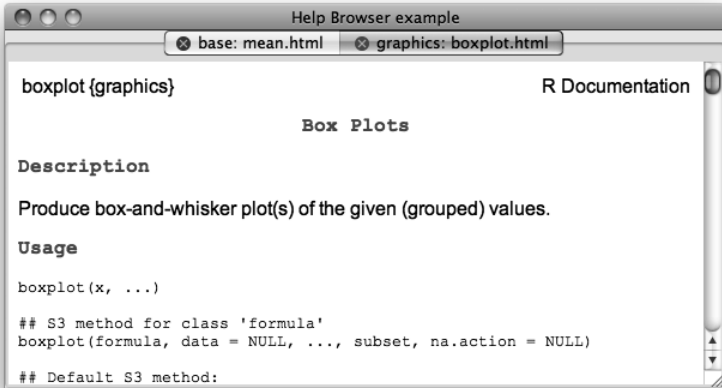


Figure 13.3: An example in which a notebook is used to display various help pages shown in a `QWebView` instance.

```
options("browser" = function(url) openPage(url))
})
```

The lone new method for this class is one called to open a page. The `url` value is generated by R's help system.

```
qsetMethod("openPage", HelpBrowser, function(url) {
  tokens <- strsplit(url, "/")[1]
  tab_title <- sprintf("%s: %s", tokens[length(tokens)-2],
                      tokens[length(tokens)])
  webview <- Qt$QWebView()
  webview$setUrl(Qt$QUrl(url))
  this$currentIndex <- addTab(webview, tab_title)
})
```

Figure 13.3 was created through this invocation:

```
help_browser <- HelpBrowser()
help_browser$windowTitle <- "Help Browser example"
help_browser$show()
help_browser$raise()
##
options("help_type"="html")
help("mean")
help("boxplot")
```

General widget stacking It is sometimes useful to have a widget that shows only one of its widgets at once, like a `QTabWidget`, except without

the tabs. There is no way to hide the tabs of `QTabWidget`. Instead, we should use `QStackedWidget`, which stacks its children so that only the widget on top of the stack is visible. There is no way for the user to switch between children; it must be done programmatically. The actual layout is managed by `QStackedLayout`, which should be used directly if only a layout is needed, e.g., as a sub-layout.

13.8 Scroll areas

Sometimes a widget is too large to fit in a layout and thus must be displayed partially. Scroll bars then allow the user to adjust the visible portion of the widget. Widgets that often become too large include tables, lists, and text-edit panes. These inherit from `QAbstractScrolledArea` and thus natively provide scroll bars without any special attention from the user. Occasionally, we are dealing with a widget that lacks such support and thus need to explicitly embed the widget in a `QScrollArea`. This often arises when displaying graphics and images. To demonstrate, we will create a simple zoomable image viewer. The user can zoom in and out and use the scroll bars to pan around the image. First, we place an image in a label and add it to a scroll area:

```
image <- Qt$QLabel()
image$ixmap <- Qt$QPixmap("someimage.png")
scroll_area <- Qt$QScrollArea()
scroll_area$setWidget(image)
```

Next, we define a function for zooming in on the image:

```
zoomImage <- function(x = 2.0) {
  image$resize(x * image$ixmap$size())
  updateScrollBar <- function(sb) {
    sb$value <- x * sb$value + (x - 1) * sb$pageStep / 2
  }
  updateScrollBar(scroll_area$horizontalScrollBar())
  updateScrollBar(scroll_area$verticalScrollBar())
}
```

Of note here is that we are scaling the size of the pixmap using the `*` function, which `qtbase` is forwarding to the corresponding method on the `QSize` object. Updating the scroll bars is also somewhat tricky, since their value corresponds to the top left, while we want to preserve the center point. We leave the interface for calling the `zoomImage` function as an exercise for the interested reader.

The geometry of a scroll area is such that there is an empty space in the corner between the ends of the scroll bars. To place a widget in the corner, pass it to the `setCornerWidget` method.

13.9 Paned windows

`QSplitter` is a split-pane widget, a container that splits its space between its children, with draggable separators that adjust the balance of the space allocation.

Unlike `GtkPaned` in `GTK+`, there is no limit on the number of child panes. We add three with `addWidget`:

```
splitter <- Qt$QSplitter()
splitter$addWidget(Qt$QLabel("One"))
splitter$addWidget(Qt$QLabel("Two"))
splitter$addWidget(Qt$QLabel("Three"))
```

The orientation can be adjusted dynamically through `setOrientation`.

```
splitter$setOrientation(Qt$Qt$Vertical)
```

In addition to adjusting the space allocation with a mouse, we can adjust the sizes programmatically through the `setSizes` method:

```
splitter$setSizes(c(100L, 200L, 300L))
```

If needed, we can connect to the `splitterMoved` signal. The callback receives the position of the moved handle and its index.

This chapter covers some of the basic dialogs and widgets provided by Qt. Together with layouts, these form the basis for most user interfaces. The next chapter will introduce the more complex widgets that typically act as views for separate data models.

14.1 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 and wizards. 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 provides two ways to create a message box (Figure 14.1). 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 call will block the flow of the program until the user responds and returns the standard identifier for the button that was clicked. Each type

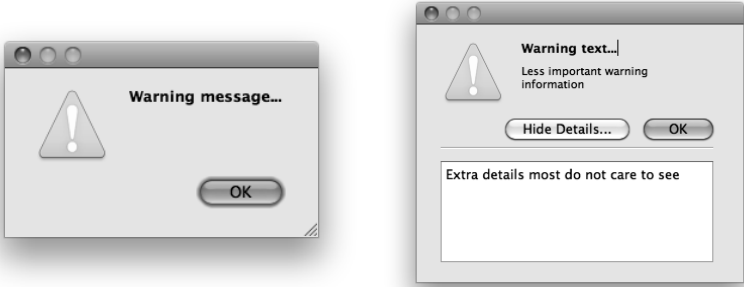


Figure 14.1: Message dialog boxes. The left one was made with the convenient static method, the right – with more detail – using `QMessageBox` methods.

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 he or she responds. If the parent is `NULL`, as in this case, input to all windows is blocked. Specifying the parent will automatically position the dialog 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 may take a more gradual path. Here, we construct an instance of `QMessageBox`. It is possible to specify several properties at construction. The following is how we might construct a warning dialog:

```
dialog <- 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`:

```
dialog$informativeText <- "Less important warning information"
```

```
dialog$detailedText <- "Extra details most do not care to see"
```

Both provide additional textual information at an increasing level of detail. The `informativeText` 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 we specify 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.

```
dialog$exec() # returns response code
```

```
[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:

```
qconnect(dialog, "finished", function(response) {
  dialog$close()
})
```

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.

Finally, we show, raise, and activate the dialog with:

```
dialog$show()
dialog$raise()
dialog$activateWindow()
```

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:

```
dialog <- Qt$QMessageBox()
dialog$windowTitle <- "[This space for rent]"
dialog$text <- "This is the main text"
dialog$informativeText <- "This should give extra info"
dialog$detailedText <- "And this provides\neven more detail"
dialog$icon <- Qt$QMessageBox$Critical
dialog$standardButtons <-
  Qt$QMessageBox$Cancel | Qt$QMessageBox$Ok
## 'Cancel' instead of 'Ok' is the default
```

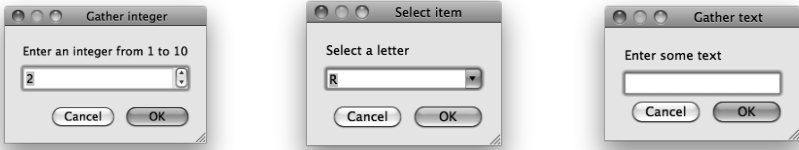


Figure 14.2: Qt provides three static constructors for input dialogs, making it straightforward to collect integers, selections, or text from a user.

```
dialog$setDefaultButton(Qt$QMessageBox$Cancel)
##
if(dialog$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” (Figure 14.2).

Like with `QMessageBox`, we 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 us 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:

```
even_integer <- 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:



Figure 14.3: Dialog button boxes and their implementation under Mac OS X and Linux.

```
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 by 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.

```
dialog <- Qt$QInputDialog()
dialog$setWindowTitle("Select item")
dialog$setLabelText("Select a letter")
dialog$setComboBoxItems(LETTERS)
dialog$setTextValue(LETTERS[18])
dialog$options(Qt$QInputDialog$UseListViewForComboBoxItems)
```

```
if (dialog$exec())
  print(dialog$textValue())
```

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

Button boxes

Before discussing custom dialogs, we first introduce the `QDialogButtonBox` utility for arranging dialog buttons in a consistent and cross-platform manner. Dialogs often have a standard button placement that varies among desktop environments. `QDialogButtonBox` is a container of buttons that arranges its children according to the convention of the platform. We place some standard buttons into a button box:

```
btn_box <- Qt$QDialogButtonBox(Qt$QDialogButtonBox$Ok |
                               Qt$QDialogButtonBox$Cancel |
                               Qt$QDialogButtonBox$Help)
```

Figure 14.3 shows how the buttons are displayed on two different operating systems. To indicate the desired buttons, we pass a combination of flags from the `QDialogButtonBox$StandardButton` enumeration. Each standard button code implies a default label and role, taken from the `QDialogButtonBox$ButtonRole` enumeration. In the above example, we added a standard `Ok` button, with the label “Ok” (depending on the language) and the role `AcceptRole`. The `Cancel` button has the appropriate label and `CancelRole` as its role. Icons are also displayed, depending on the platform and theme. The benefits of using standard buttons include convenience, standardization, platform consistency, and automatic translation of labels.

To respond to user input, we can connect directly to the clicked signal on a given button. It is often more convenient, however, to connect to one of the high-level button-box signals, which include: `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 the button object.

```
qconnect(btn_box, "accepted", function() message("accepted"))
qconnect(btn_box, "rejected", function() message("rejected"))
qconnect(btn_box, "helpRequested", function() message("help"))
qconnect(btn_box, "clicked",
         function(button) message(button$text))
```

The first button added with the `AcceptRole` role is made the default. Overriding this requires adding the default button with `addButton` and setting the default property on the returned button object.

Custom dialogs

Every dialog in Qt inherits from `QDialog`, which we can leverage for our own custom dialogs. One approach is to construct an instance of `QDialog` and add arbitrary widgets to its layout. However, we suggest an alternative approach: extend `QDialog` or one of its derivatives and implement the custom functionality in a subclass. This more formally encapsulates the state and behavior of the custom dialog. We demonstrate the subclass approach by constructing a dialog that requests a date from the user.

We begin by defining our class and its constructor:

```
qsetClass("DateDialog", Qt$QDialog,
         function(parent = NULL) {
           super(parent=parent)
           setWindowTitle("Choose a date")
         })
```



Figure 14.4: A custom dialog, embedding a date-selection widget with a `QDialog` instance.

```

this$calendar <- Qt$QCalendarWidget ()
#
btn_box <-
  Qt$QDialogButtonBox(Qt$QMessageBox$Cancel |
                      Qt$QMessageBox$Ok)
qconnect(btn_box, "accepted", function() {
  this$close()
  this$setResult(Qt$QMessageBox$Ok)
})
qconnect(btn_box, "rejected",
         function() this$close())
#
layout <- Qt$QVBoxLayout ()
sapply(list(calendar, btn_box), layout$addWidget)
setLayout(layout)
})

```

Our dialog consists of a calendar, implemented by the `QCalendarWidget`, and a set of response buttons, organized by a `QDialogButtonBox`. The calendar is stored as a field on the instance, so that we can retrieve the selected date upon request.

We define a method that gets the currently selected date:

```

qsetMethod("selectedDate", DateDialog,
          function(x) calendar$selectedDate$toString())

```

`DateDialog` can be executed like any other `QDialog`:

```

date_dialog <- DateDialog()
if (date_dialog$exec())
  message(date_dialog$selectedDate())

```

Wizards

QWizard implements a wizard – a multipage dialog that guides the user through a sequential, possibly branching process. Wizards are composed of pages, and each page has a consistent interface, usually including buttons for moving backward and forward through the pages. The look and feel of a QWizard is consistent with platform conventions.

We create a wizard object and set its title:

```
wizard <- Qt$QWizard()  
wizard$setWindowTitle("A wizard")
```

Each page is represented by a QWizardPage. We create one for requesting the age of the user and add the page to the wizard:

```
get_age_page <- Qt$QWizardPage(wizard)  
get_age_page$setTitle("What is your age?")  
layout <- Qt$QFormLayout()  
get_age_page$setLayout(layout)  
layout$addRow("Age", (age <- Qt$QLineEdit()))  
wizard$addPage(get_age_page)
```

Two more pages are added:

```
get_toys_page <- Qt$QWizardPage(wizard)  
get_toys_page$setTitle("What toys do you like?")  
layout <- Qt$QFormLayout()  
get_toys_page$setLayout(layout)  
layout$addRow("Toys", (toys <- Qt$QLineEdit()))  
wizard$addPage(get_toys_page)  
##  
get_games_page <- Qt$QWizardPage(wizard)  
get_games_page$setTitle("What games do you like?")  
layout <- Qt$QFormLayout()  
get_games_page$setLayout(layout)  
layout$addRow("Games", (games <- Qt$QLineEdit()))  
wizard$addPage(get_games_page)
```

Finally, we run the wizard by calling its exec method:

```
response <- wizard$exec()  
if(response)  
  message(toys$text)
```

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:

```
filename <- Qt$QFileDialog$getOpenFileName(NULL,
                                           "Open a file...", getwd())
```

All take as initial arguments a parent, a caption and a directory. Other arguments allow us to set a filter, say. For basic use, these are nearly as easy to use as R's `file.choose` function. If a file is selected, `filename` will contain the full path to the file; otherwise it will be `NULL`.

To allow multiple selection, call the plural form of the method:

```
filenames <- Qt$QFileDialog$getOpenFileNames(NULL,
                                              "Open file(s)...", getwd())
```

To select a file name for saving, we have:

```
filename <- 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:

```
name_filter <- paste("R files (*.R .RData)",
                   "Sweave files (*.Rnw)",
                   "All files (*.*)",
                   sep=";;")
##
filenames <- Qt$QFileDialog$getOpenFileNames(NULL,
                                             "Open file(s)...", getwd(), name_filter)
```

Although the static functions provide most of the functionality, to fully configure a dialog, it may be necessary to construct and manipulate a dialog instance explicitly. 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 such as permissions.

Example 14.1: File dialogs

We construct a dialog for opening an R-related file, using the directory name selected above as the history:

```
dialog <- Qt$QFileDialog(NULL, "Choose an R file", getwd(),
                        name_filter)
```



```
dialog$fileMode <- Qt$QFileDialog$ExistingFiles
dialog$setHistory(dirname)
```

The dialog is executed like any other. To get the specified files, call `selectedFiles`:

```
if(dialog$exec())
  print(dialog$selectedFiles())
```

Other choosers

Qt provides several additional dialog types for choosing a particular type of item. These include `QColorDialog` for picking a color and `QFontDialog` for selecting a font. These special-case dialogs will not be discussed further here.

14.2 Labels

As demonstrated in many of the preceding examples, 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:

```
label <- Qt$QLabel("<font color='red'>Red</font>")
```

By default, `QLabel` guesses whether the string is rich or plain text. In the above, the rich-text format is identified from the markup. The `textFormat` property can override this.

The label text is stored in the `text` property. Properties relevant to text layout include `alignment`, `indent` (in pixels), `margin`, and `wordWrap`.

14.3 Buttons

As we have seen, the ordinary button in Qt is created by `QPushButton`, which inherits most of its functionality from `QAbstractButton`, the common base class for buttons. We create a simple “Ok” button:

```
button <- Qt$QPushButton("Ok")
```

Like any other widget, a button can be disabled, so that the user cannot press it:

```
button$enabled <- FALSE
```

This is useful for preventing the user from attempting to execute commands that do not apply to the current state of the application. Qt changes the rendering widget, including that of the icon, to indicate the disabled state.

Signals A push button usually executes some command when clicked or otherwise invoked. The `QAbstractButton` class provides the signals `clicked`, for when the button is activated, and `pressed` and `released` to track button clicks and releases. For example, to respond with a simple message, we could have:

```
qconnect(button, "clicked", function() message("Ok clicked"))
```

Icons and pixmaps

A button is often decorated with an icon, which serves as a visual indicator of the purpose of the button. The `QIcon` class represents an icon. Icons can be defined for different sizes and display modes (normal, disabled, active, selected); however, this is often not necessary, as Qt will adapt an icon as necessary. As we have seen, Qt automatically adds the appropriate icon to a standard button in a dialog. When using `QPushButton` directly, there are no such conveniences. For our “Ok” button, we could add an icon from a file:

```
icon_file <- system.file("images/ok.gif", package="gWidgets")
button$icon <- Qt$QIcon(icon_file)
```

However, in general, this will not be consistent with the current style. Instead, we need to get the icon from the `QStyle`:

```
style <- Qt$QApplication$style()
button$icon <- style$standardIcon(Qt$QStyle$SP_DialogOkButton)
```

The `QStyle::StandardPixmap` enumeration lists all of the possible icons that a style should provide. In the above, we passed the key for an “Ok” button in a dialog.

We can also create a `QIcon` from image data in a `QPixmap` object. `QPixmap` stores an image in a manner that is efficient for display on the screen.¹ We can load a pixmap from a file or create a blank image and draw on it using the Qt painting API (not discussed in this book). Also, using the `qtutils` package, we can draw a pixmap using the R graphics engine. For example, the following uses `ggplot2` to generate an icon representing a histogram. First, we create the Qt graphics device (cf. Section 14.10) and plot the icon with `grid`:

```
require(qtutils)
device <- QT()
grid::grid.newpage()
grid::grid.draw(ggplot2::GeomHistogram$icon())
```

¹`QPixmap` is not to be confused with `QImage`, which is optimized for image manipulation, or the vector-based `QPicture`.

Next, we create the blank pixmap and render the device to a paint context attached to the pixmap:

```
pixmap <- Qt$QPixmap(device$size$toSize())
pixmap$fill()
painter <- Qt$QPainter()
painter$begin(pixmap)
device$render(painter)
painter$end()
```

Finally, we use the icon in a button:

```
button <- Qt$QPushButton("Histogram")
button$setIcon(Qt$QIcon(pixmap))
```

14.4 Checkboxes

The `QCheckBox` class implements a checkbox. Like the `QPushButton` class, `QCheckBox` extends `QAbstractButton`. Thus, `QCheckBox` inherits the signals `clicked`, `pressed`, and `released` and the signal `stateChanged` is added.

We create a checkbox for demonstration with:

```
checkbox <- Qt$QCheckBox("Option")
```

The `checked` property indicates whether the button is checked:

```
checkbox$checked
```

```
[1] FALSE
```

Sometimes, it is useful for a checkbox to have an indeterminate state that is neither checked nor unchecked. To enable this, set the `tristate` property to `TRUE`. In that case, we need to call the `checkState` method to determine the state, as it is no longer Boolean but from the `Qt::CheckState` enumeration.

The `stateChanged` signal is emitted whenever the checked state of the button changes:

```
qconnect(checkbox, "stateChanged", function(state) {
  if (state == Qt$Qt$Checked)
    message("checked")
})
```

The argument is from the `Qt::CheckState` enumeration; it is not a logical vector.

Groups of checkboxes

Checkboxes and other types of buttons are often naturally grouped into logical units. The frame widget, `QGroupBox`, is appropriate for visually

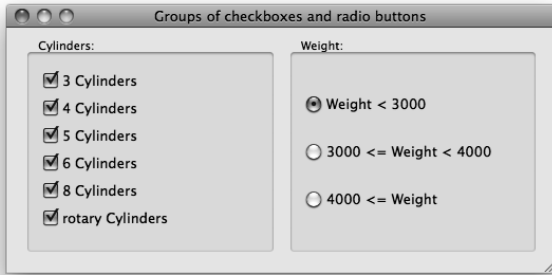


Figure 14.5: Screenshot of checkboxes and radio buttons, grouped using a `QGroupBox` instance.

representing this grouping. However, `QGroupBox` holds any type of widget, so it has no high-level notion of a group of buttons. The `QButtonGroup` object, which is *not* a widget, fills this gap by formalizing the grouping of buttons behind the scenes.

To demonstrate (Figure 14.5), we will construct an interface for filtering a data set by the levels of a factor. A common design is to have each factor level correspond to a check button in a group. For our example, we take the `cylinders` variable from the `Cars93` data set of the `MASS` package. First, we create our `QGroupBox` as the container for our buttons:

```
window <- Qt$QWidget ()
group_box <- Qt$QGroupBox ("Cylinders:")
layout <- Qt$QVBoxLayout ()
window$setLayout (layout)
```

Next, we create the button group:

```
btn_group <- Qt$QButtonGroup ()
btn_group$exclusive <- FALSE
```

By default, the buttons are exclusive, as in a radio button group. We disable that above by setting the `exclusive` property to `"FALSE"`.

We add a button for each level of the `"Cylinders"` variable to both the button group and the layout of the group box widget:

```
data(Cars93, package="MASS")
cylinders <- levels(Cars93$Cylinders)
sapply(seq_along(cylinders), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cylinders[i]))
  layout$addWidget(button)
  btn_group$addButton(button, i)
})
```

```
sapply(btn_group$buttons(),
       function(button) button$checked <- TRUE)
```

Every button is initially checked. (The `buttons` method returns a list of the managed buttons.)

Buttons can be removed through `removeButton`, where the button object (not its index) is specified for removal.

Here, we retrieve the buttons in the group and query their checked state:

```
checked <- sapply(btn_group$buttons(), function(i) i$checked)
if(any(checked)) {
  checked_cyls <- Cars93$Cylinders %in% cylinders[checked]
  message(sprintf("You've selected %d cases",
                  sum(checked_cyls)))
}
```

Button groups emit signals paralleling the `QAbstractButton` class (in particular the `buttonClicked` signal, but also `buttonPressed` and `buttonReleased`). By attaching a callback to the `buttonClicked` signal,² we will be informed when any of the buttons in the group are clicked:

```
qconnect(btn_group, "buttonClicked(QAbstractButton*)",
         function(button) {
           msg <- sprintf("Level '%s': %s",
                          button$text, button$checked)
           message(msg)
         })
```

14.5 Radio groups

Another type of checkable button is the radio button, `QRadioButton`. Radio buttons always belong to a group, and only one radio button in a group may be checked at once (they are exclusive). Continuing our filtering example (Figure 14.5), we create several radio buttons for choosing a range for the "Weight" variable in the "Cars93" data set:

```
window <- Qt$QGroupBox("Weight:")
radio_buttons <-
  list(Qt$QRadioButton("Weight < 3000", w),
       Qt$QRadioButton("3000 <= Weight < 4000", w),
       Qt$QRadioButton("4000 <= Weight", w))
```

In the above we specified the parent to the constructor to group the objects.

The simplest way to arrange the radio boxes is to place them into the same layout:

²See Section 12.6 for why we need the `(QAbstractButton*)`.

```
layout <- Qt$QVBoxLayout()
window$setLayout(layout)
sapply(radio_buttons, layout$addWidget)
radio_buttons[[1]]$setChecked(TRUE)
```

As with any other derivative of `QAbstractButton`, the checked state is stored in the `checked` property:

```
radio_buttons[[1]]$checked
```

```
[1] TRUE
```

The button's toggled signal is emitted when a button is checked or unchecked:

```
sapply(radio_buttons, function(button) {
  qconnect(button, "toggled", function(checked) {
    if(checked) {
      message(sprintf("You checked %s.", button$text))
    }
  })
})
```

Managing the radio buttons in a list, as above, is often inconvenient and difficult to maintain. Instead, we can have a `QButtonGroup` instance manage the radio buttons:

```
btn_group <- Qt$QButtonGroup()
lapply(radio_buttons, btn_group$addButton)
```

Since our button group is exclusive, we can query for the currently checked button through the `checkedButton` method:

```
btn_group$checkedButton()$text
```

```
[1] "Weight < 3000"
```

As well, we can listen for events on the button group, rather than listen on each radio button, as was done above. This strategy makes it much easier to add (or remove) items, although we do need to add to (or remove from) both the layout and the button group.

14.6 Combo boxes

A combo box allows a single selection from a drop-down list of options. In this section, we describe the basic usage of `QComboBox`. This includes populating the menu with a list of strings and optionally allowing arbitrary input through an associated text entry. For the more complex approach of deriving the menu from a separate data model, see Section 15.3.

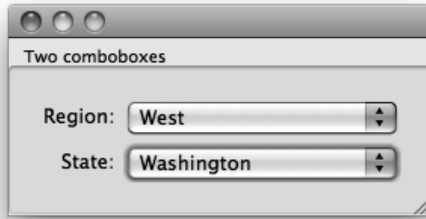


Figure 14.6: Two combo boxes in a form layout.

This example shows how one combo box, listing regions in the United States, updates another, which lists states in that region (Figure 14.6). First, we prepare a `data.frame` with the name, region, and population of each state and split that `data.frame` by the regions:

```
df <- data.frame(name=state.name, region=state.region,
                 population=state.x77[, 'Population'],
                 stringsAsFactors=FALSE)
states_by_region <- split(df, df$region)
```

We create our combo boxes, loading the region combo box with the regions:

```
state_combo <- Qt$QComboBox()
region_combo <- Qt$QComboBox()
region_combo$addItem(names(states_by_region))
```

The `addItem` method accepts a character vector of options and is the most convenient way to populate a combo box with a simple list of strings.

To retrieve the value, the `currentText` property holds the current text, whereas the `currentIndex` property indicates the index of the currently selected item:

```
region_combo$currentText
```

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

```
region_combo$currentIndex # 0-based
```

```
[1] 0
```

By setting it to `-1`, we clear the selection.

```
region_combo$currentIndex <- -1
```

To respond to a change in the current index, we connect to the activated signal:

```
qconnect(region_combo, "activated(int)", function(index) {
  state_combo$clear()
  state_combo$addItem(states_by_region[[index+1]]$name)
})
```

Our handler resets the state combo box to correspond to the selected region, indicated by "index", which is 0-based.

Finally, we place the widgets in a form layout:

```
window <- Qt$QGroupBox("Two combo boxes")
layout <- Qt$QFormLayout()
window$setLayout(layout)
layout$addRow("Region:", region_combo)
layout$addRow("State:", state_combo)
layout$fieldGrowthPolicy <- # grow combo boxes
Qt$QFormLayout$AllNonFixedFieldsGrow
```

To allow a user to enter a value not in the menu, the property `editable` can be set to `TRUE`. This would not be sensible for our example.

14.7 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 spin boxes offer a discrete choice. However, underlying each is an arithmetic sequence. Our example will include both widgets and synchronize them for specifying a single range. The slider allows for quick movement across the range, while the spin box is best suited for fine adjustments.

Sliders

Sliders are implemented by `QSlider`, a subclass of `QAbstractSlider`. The class allows selection only from integer values. We create an instance and specify the bounds of the range:

```
slider <- Qt$QSlider()
slider$minimum <- 0
slider$maximum <- 100
```

We can also customize the step size:

```
slider$singleStep <- 1
slider$pageStep <- 5
```

Single step refers to the effect of pressing one of the arrow keys, while pressing "Page Up/Down" adjusts the slider by `pageStep`.

The current cursor position is given by the property value; we set it to the middle of the range:

```
slider$value
```

```
[1] 0
```

```
slider$value <- 50
```

A slider has several aesthetic properties. We set our slider to be oriented horizontally (vertical is the default) and place the tick marks below the slider, with a mark every ten values:

```
slider$orientation <- Qt$Qt$Horizontal
slider$tickPosition <- Qt$QSlider$TicksBelow
slider$tickInterval <- 10
```

The `valueChanged` signal is emitted whenever the `value` property is modified. An example is given below, after the introduction of the spin box.

Spin boxes

There are several spin-box classes: `QSpinBox` (for integers), `QDoubleSpinBox` and `QDateTimeEdit`. All of these derive from a common base, `QAbstractSpinBox`. As our slider is integer-valued, we will introduce `QSpinBox` here. Configuring a `QSpinBox` proceeds much as it does for `QSlider`:

```
spinbox <- Qt$QSpinBox()
spinbox$minimum <- slider$minimum
spinbox$maximum <- slider$maximum
spinbox$singleStep <- slider$singleStep
```

There is no `"pageStep"` for a spin box. Since we are communicating a percentage, we specify `"%"` as the suffix for the text of the spin box:

```
spinbox$suffix <- "%"
```

It is also possible to set a prefix.

Both `QSlider` and `QSpinBox` emit the `valueChanged` signal whenever the value changes. We connect to the signal on both widgets to keep them synchronized:

```
f <- function(value, obj) obj$value <- value
qconnect(spinbox, "valueChanged", f, user.data = slider)
qconnect(slider, "valueChanged", f, user.data = spinbox)
```

We pass the other widget as the user data, so that state changes in one are forwarded to the other. A race condition is avoided, as `valueChanged` is emitted only when the value actually changes.

14.8 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:

```
edit <- Qt$QLineEdit("Initial contents")
```

The `text` property holds the current value:

```
edit$text
```

```
[1] "Initial contents"
```

Here we select the text, so that the initial contents are overwritten when the user begins typing:

```
edit$setSelection(start = 0, length = nchar(edit$text))
```

```
edit$selectedText
```

```
[1] "Initial contents"
```

If `dragEnabled` is `TRUE`, the selected text can be dragged and dropped on the appropriate targets.

By default, the line edit displays the typed characters. Other echo modes are available, as specified by the `echoMode` property. For example, the `Qt$QLineEdit$Password` mode will behave as a password entry, echoing only asterisks.

In Qt versions 4.7 and above, we can specify place-holder text that fills the entry if it is empty and unfocused. Typically, this text indicates to the user the expected contents of the entry:

```
edit$text <- ""
edit$setPlaceholderText("Enter some text here")
```

The `editingFinished` signal is emitted when the user has committed the edit, typically by pressing the return key, and the input has been validated:

```
qconnect(edit, "editingFinished", function() {
  message("Entered text: '", edit$text, "'")
})
```

To respond to any editing, without waiting for it to be committed, requires connecting to the `textEdited` signal. The newly entered text is passed to the callback.

The `selectionChanged` signal reports selection changes.

Completion

Using the `QCompleter` framework, a list of possible words can be presented for completion when text is entered into a `QLineEdit`.

Example 14.2: Implementing completion of Qt classes and methods

This example shows how completion can assist in exploring the classes and namespaces of the Qt library. A form layout arranges two line edit widgets – one to gather a class name and one for method and property names. See Figure 14.8 to see this widget example embedded into a web page.

```
class_browser <- Qt$QWidget()
layout <- Qt$QFormLayout()
class_browser$setLayout(layout)
layout$addRow("Class name", class_edit <- Qt$QLineEdit())
layout$addRow("Method name", method_edit <- Qt$QLineEdit())
```

Next, we construct the completer for the class entry, listing the components of the "Qt" environment with `ls`:

```
class_completer <- Qt$QCompleter(ls(Qt))
class_edit$setCompleter(class_completer)
```

The completion for the methods depends on the class. As such, we update the completion when editing is finished for the class name:

```
qconnect(class_edit, "editingFinished", function() {
  class_name <- class_edit$text
  if(class_name == "") return()
  class_object <- get(class_name, envir = Qt)
  if(!is.null(class_object)) {
    method_completer <- Qt$QCompleter(ls(class_object()))
    method_edit$setCompleter(method_completer)
  }
})
```

Masks and validation

`QLineEdit` has various means to restrict and validate user input. The `maxLength` property restricts the number of allowed characters. Beyond that, there are two mechanisms for validating input: masks and `QValidator`. An input mask is convenient for restricting input to a simple pattern. We could, for example, force the input to conform to the pattern of a United States Social Security number:

```
edit$inputMask <- "999-99-9999"
```

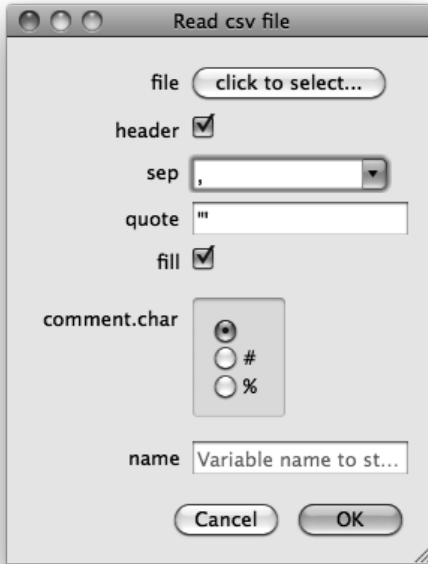


Figure 14.7: A dialog to collect argument for a call to `read.csv`.

Please see the API documentation of `QLineEdit` for a full description of the format of an input mask.

As illustrated in Example 12.2, `QValidator` is a much more general validation mechanism, where the value in the widget is checked by the validator before being committed.

Example 14.3: A dialog for calling `read.csv`

We illustrate some of the widgets and dialogs discussed in this chapter in the following example, which gathers arguments needed to import a file into R through `read.csv`. Figure 14.7 shows the finished GUI. We use a form layout to organize our controls, but first we need to define them.

We use a named list below to store our controls:

```
controls <- list()
controls$file <- Qt$QPushButton("click to select...")
##
controls$header <- Qt$QCheckBox() # no name
controls$header$setChecked(TRUE)
##
controls$sep <- Qt$QComboBox()
controls$sep$addItem(sprintf('%s', c(",",";","","\t")))
```

```
controls$sep$setEditable(TRUE)
##
controls$quote <- Qt$QLineEdit("\'")
##
controls$fill <- Qt$QCheckBox()
controls$fill$setChecked(TRUE)
```

The names of the list will become the label associated with the corresponding control. A button is chosen for the file, which we will later use to open a file selection dialog. Otherwise, the controls have a fairly obvious mapping to the arguments of `read.csv`.

To illustrate radio buttons, we use a set of them to select the comment character argument. Here we store the container in the list and create a separate (global) variable to hold the radio-button widgets themselves.

```
controls$comment.char <- Qt$QGroupBox() # container
comment.char <- lapply(sprintf("%s", c("","#", "%")),
                       Qt$QRadioButton, controls$comment.char)
comment.char[[1]]$setChecked(TRUE)
## manage
comment.char.bg <- Qt$QButtonGroup()
sapply(comment.char, comment.char.bg$addButton)
## layout
layout <- Qt$QVBoxLayout()
controls$comment.char$setLayout(layout)
sapply(comment.char, layout$addWidget)
```

The variable name uses a simple line-edit widget to which we add an instructional placeholder. We also populate its auto-completion database with the current global workspace variable names.

```
controls$name <- Qt$QLineEdit("")
controls$name$setPlaceholderText("Variable name to store data")
completer <- Qt$QCompleter(ls(.GlobalEnv))
controls$name$setCompleter(completer)
```

The form layout goes quickly, as we can iterate over the list components:

```
form_layout <- Qt$QFormLayout()
mapply(form_layout$addRow, names(controls), controls)
```

A dialog button box ensures consistency with the operating -system conventions.

```
button_box <-
  Qt$QDialogButtonBox(Qt$QMessageBox$Cancel |
                      Qt$QMessageBox$Ok)
```

We use a simple widget to lay out the form and the buttons.

```

window <- Qt$QWidget()
window$windowTitle <- "Read csv file"
window$setLayout(window_layout <- Qt$QVBoxLayout())
window_layout$addLayout(form_layout)
window_layout$addWidget(button_box)

```

At this point, the widgets are set up and laid out. We turn to the task of adding interactivity. First, the file button, when clicked, should open a file-selection dialog. If a file load is successful, we change the label on the button to indicate the selection, using the global filename to store the value.

```

filename <- NULL
qconnect(controls$file, "clicked", function() {
  name_filter <- "CSV file (*.csv);; All files (*.*)"
  filename <<- Qt$QFileDialog$getOpenFileName(window,
    "Select a CSV file...", getwd(), name_filter)
  if(!is.null(filename))
    controls$file$setText(basename(filename))
})

```

We connect to the signals on the dialog button box. The rejected callback simply hides the dialog. The accepted callback is more complex. After checking that a file and variable name have been selected, we gather the values from the dialog through various means. These are stored in the list args below. Finally, once the arguments are collected, we execute the call to read.csv.

```

qconnect(button_box, "rejected", function() window$hide())
##
qconnect(button_box, "accepted", function() {
  if(!is.null(filename) && nchar(controls$name$text) > 0) {
    args <- list(file=filename,
      header=controls$header$checked,
      sep=controls$sep$currentText,
      quote=controls$quote$text,
      fill=controls$fill$checked
    )
    args$comment.char <- comment.char.bg$checkedButton()$text
    ##
    val <- do.call("read.csv", args)
    assign(controls$name$text, val, .GlobalEnv)
    window$hide()
  } else {
    Qt$QMessageBox$warning(parent = window,
      title = "Warning!",
      text = "You need to select a file and variable name")
  }
})

```

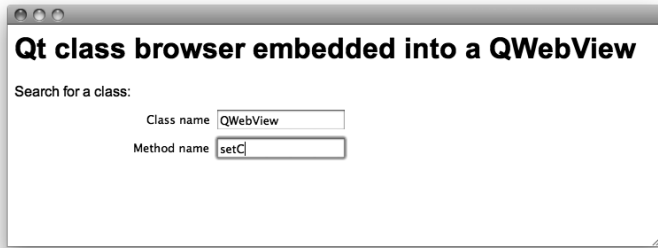


Figure 14.8: An example of QWebView holding an embedded widget within a web page.

14.9 QWebView widget

The QtWebKit module provides a Qt-based implementation of the cross-platform WebKit API. The standards support is comparable to that of other WebKit implementations like Safari and Chrome. This includes HTML version 5, Javascript and SVG. The Javascript engine, provided by the QtScript module, allows bridging Javascript and R, which will not be discussed. The widget QWebView uses QtWebKit to render web pages in a GUI.

This is the basic usage:

```
webview <- Qt$QWebView()
webview$load(Qt$QUrl("http://www.r-project.org"))
```

A web browser typically provides feedback on the URL loading process. The signals `loadStarted`, `loadProgress`, and `loadFinished` are provided for this purpose. History information is stored in a `QWebHistory` object, retrieved by calling `history` on the web view. This could be used for implementing a “Back” button.

Embedding Qt widgets A unique feature of QtWebKit is the ability to embed Qt widgets into a web page (Figure 14.8). This is one mechanism for constructing hybrid web/desktop applications. Widget embedding is implemented through the standard HTML “object” tag. We can register a plug-in, manifested as a `QWidget`, for a particular MIME type, specified through the “type” attribute of the “object” element.

For example, we might have the following simple HTML:

```
html <- readLines(out <- textConnection("
<html xmlns='http://www.w3.org/1999/xhtml'>
  <body>
    <h1>Qt class browser embedded into a QWebView</h1>
    Search for a class:<br/>
```

```

    <object type='application/x-qt-class-browser' width='500'
        height='100' />
  </body>
</html>
"); close(out)
html <- paste(html, collapse = "\n")

```

For our plug-in, we use the class browser widget, constructed in Example 14.2. To provide the plug-in, we need to implement a custom `QWebPluginFactory`:

```
qsetClass("RPluginFactory", Qt$QWebPluginFactory)
```

The factory has two duties: describing its available plug-ins and creating a plug-in, in the form of a `QWidget`, for a given MIME type. The `plugins` method returns a list of plug-in descriptions:

```

qsetMethod("plugins", RPluginFactory, function() {
  plugin <- Qt$QWebPluginFactory$Plugin()
  plugin$setName("Class Browser")
  mimeType <- Qt$QWebPluginFactory$MimeType()
  mimeType$setName("application/x-qt-class-browser")
  plugin$setMimeTypes(list(mimeType))
  list(plugin)
})

```

Our factory provides a single plug-in, with a single MIME type that matches the type of the "object" element in the HTML. The `create` method constructs the actual `QWidget` corresponding to the plug-in:

```

qsetMethod("create", RPluginFactory,
  function(mime_type, url, arg_names, arg_vals) {
    if (mime_type == "application/x-qt-class-browser")
      class_browser
    else Qt$QWidget()
  })

```

If the MIME type does not match our plug-in, we simply return an empty widget.

Finally, we need to enable plug-ins, register our factory, and load the HTML:

```

globalSettings <- Qt$QWebSettings$globalSettings()
globalSettings$setAttribute(Qt$QWebSettings$PluginsEnabled,
                             TRUE)
webview$page()$setPluginFactory(RPluginFactory())
webview$setHtml(html)

```


14.10 Embedding R graphics

The `qtutils` package includes a Qt-based graphics device, written by Deepayan Sarkar. We make a simple scatterplot:

```
library(qtutils)
qt_device <- QT()
plot(mpg ~ hp, data = mtcars)
```

The `"qtDevice"` object may be shown directly or embedded within a GUI. For example, we might place it in a notebook of multiple plots:

```
notebook <- Qt$QTabWidget()
notebook$addTab(qt_device, "Plot 1")
print(notebook)
```

The device provides a context menu with actions for zooming, exporting and printing the plot. We can execute an action programmatically by extracting the action from `"qtDevice"` and activating it.

To increase performance at a slight cost of quality, we could direct the device to leverage hardware acceleration through OpenGL. This requires passing `"opengl = TRUE"` to the QT constructor:

```
qt_opengl_device <- QT(opengl = TRUE)
```

Even without the help of OpenGL, the device is faster than most other graphics devices, in particular `cairoDevice`, due to the general efficiency of Qt graphics.

Internally, the device renders to a `QGraphicsScene`. Every primitive drawn by R becomes an object in the scene. Nothing is rasterized to pixels until the scene is displayed on the screen. This presents the interesting possibility of programmatically manipulating the graphical primitives after they have been plotted; however, this is beyond our scope. See Example 14.3 for a way to render the scene to an off-screen `QPixmap` for use as an icon.

14.11 Drag-and-drop

Some Qt widgets, such as those for editing text, natively support basic drag-and-drop activities. For other situations, it is necessary to program against the low-level drag-and-drop API, presented here. A drag-and-drop event consists of several stages: the user selects the object that initiates the drag event, drags the object to a target, and finally drops the object on the target. For our example, we will enable the dragging of text from one label to another, following the Qt tutorial. Example 15.2 has a more realistic example.

Initiating a drag

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)
  })
```

When a drag-and-drop sequence is initiated, the source, i.e., the widget receiving the mouse press event, needs to encode a chosen graphical object as MIME data. This might be as an image, text, or other data type. This occurs in the `mouseEventHandler` of the source:

```
qsetMethod("mousePressEvent", DragLabel, function(event) {
  mime_data <- Qt$QMimeData()
  mime_data$setText(text)

  drag <- Qt$QDrag(this)
  drag$setMimeData(mime_data)

  drag$exec()
})
```

We store the text in a `QMimeData` and pass it to the `QDrag` object, which represents the drag operation. The drag object is given `this` as its parent, so that drag is not garbage collected when the handler returns. Finally, calling the `exec` method is necessary to initiate the drag. It is also possible to call `setPixmap` to set a pixmap to represent the object as it is being dragged to its target.

Handling a drop

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)
    this$acceptDrops <- TRUE

    this$bgrole <- backgroundRole()
    this$alignment <- Qt$Qt$AlignCenter
```

```
        setMinimumSize(200, 200)
        this$autoFillBackground <- TRUE
        clear()
    })
```

The important step is to allow the widget to receive drops by setting `acceptDrops` to `TRUE`. The other settings ensure that the label fills a minimal amount of space and draws its background. The background role is preserved so that we can restore it later, after applying highlighting.

First, we define a couple of utility methods:

```
qsetMethod("clear", DropLabel, function() {
    setText(this$orig_text)
    setBackgroundRole(this$bgrole)
})
qsetMethod("setText", DropLabel, function(text) {
    this$orig_text <- text
    super("setText", text)           # next method
})
```

The `clear` method is used to restore the label to an initial state. The background role is remembered in the constructor, and the `setText` override saves the original text.

When the user drags an object over our target, we need to verify that the data is of an acceptable type. This is implemented by the `dragEnterEvent` handler:

```
qsetMethod("dragEnterEvent", DropLabel, function(event) {
    mime_data <- event$mimeType()
    if (mime_data$hasImage() || mime_data$hasHtml() |
        mime_data$hasText())
    {
        super("setText", "<Drop Text Here>")
        setBackgroundRole(Qt::QPalette::Highlight)
        event$acceptProposedAction()
    }
})
```

If the data type is acceptable, we accept the event. This changes the mouse cursor, indicating that a drop is possible. A secondary role of this handler is to indicate that the target is receptive to drops; we highlight the background of the label and change the text. To undo the highlighting, we override the `dragLeaveEvent` method:

```
qsetMethod("dragLeaveEvent", DropLabel, function(event) {
    clear()
})
```

Finally, we have the important drop-event handler. The following code implements this more generally than is needed for this example, as we have only text in our MIME data:

```
qsetMethod("dropEvent", DropLabel, function(event) {
  mime_data <- event$mimeTypeData()

  if(mime_data$hasImage()) {
    setPixmap(mime_data$imageData())
  } else if(mime_data$hasHtml()) {
    setText(mime_data$html)
    setTextFormat(Qt$Qt$RichText)
  } else if(mime_data$hasText()) {
    setText(mime_data$text())
    setTextFormat(Qt$Qt$PlainText)
  } else {
    setText("No match") # replace ...
  }

  setBackgroundRole(this$bgrole)
  event$acceptProposedAction()
})
```

We are passed a `QDropEvent` object, which contains the `QMimeData` set on the `QDrag` by the source. The data is extracted and translated to one or more properties of the target. The final step is to accept the drop event, so that the drag-and-drop operation is completed.

This page intentionally left blank

Qt: Widgets Using Data Models

The model, view, controller (MVC) pattern is fundamental to the design of widgets that display and manipulate data. Keeping the model separate from the view allows multiple views for the same data. Generally, the model is an abstract interface. Thus, the same view and controller components are able to operate on any data source (e.g., a database) for which a model implementation exists.

Qt provides `QAbstractItemModel` as the base for all of its data models. Like `GtkTreeModel`, `QAbstractItemModel` represents tables, optionally with a hierarchy. The precise implementation depends on the subclass. Widgets that view item models extend `QAbstractItemView` and include tables, lists, trees, and combo boxes. This section will outline the available model and view implementations in Qt and `qtbases`.

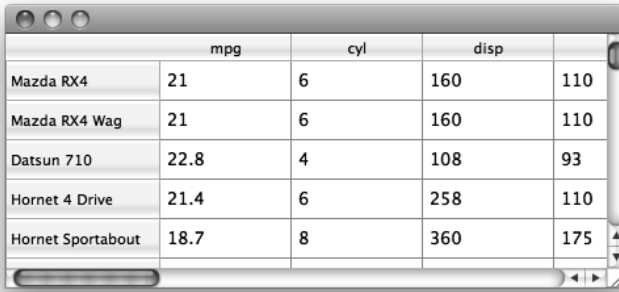
15.1 Displaying tabular data

Displaying an R data frame

As mentioned, Qt expects data to be stored in a `QAbstractItemModel` instance. In R, the canonical structure for tabular data is `data.frame`. The `DataFrameModel` class bridges these structures by wrapping `data.frame` in an implementation of `QAbstractItemModel`. This essentially allows a `data.frame` object to be passed to any part of Qt that expects tabular data. It also offers significant performance benefits: there is no need to copy the data frame into a C++ data structure, which would be especially slow if the looping occurred in R.

Displaying a simple table of data with `DataFrameModel` is much easier than with GTK+ and `RGtkDataFrame`. Here, we construct a widget to show a `data.frame` in a table view:

```
model <- qDataFrameModel(mtcars)
view <- Qt$QTableView()
view$setModel(model)
```



	mpg	cyl	disp	hp
Mazda RX4	21	6	160	110
Mazda RX4 Wag	21	6	160	110
Datsun 710	22.8	4	108	93
Hornet 4 Drive	21.4	6	258	110
Hornet Sportabout	18.7	8	360	175

Figure 15.1: Basic display of a data frame using just three commands.

Figure 15.1 shows the resulting widget. We could also pass our model to any other view expecting a `QAbstractItemModel`. For example, the first column could be displayed in a list or combo box.

The R data frame of a `DataFrameModel` can be accessed using `qdataFrame`:

```
DF <- qdataFrame(model)
DF[1:3, 1:10]
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4

Assignment is possible, too:

```
qdataFrame(model)$hpToMpg <- with(qdataFrame(model), hp / mpg)
```

Our table view now contains a new column, holding the horsepower to miles-per-gallon ratio. The `DataFrameModel` object is a reference, so modifications occur in place, rather than being incorporated in a newly constructed object. One consequence is that changes made within a function body may propagate beyond the local environment. It is important to notice that any view of the model will reflect changes to the underlying model without any explicit updating.

Headers A table view has horizontal and vertical headers. The horizontal header displays the column names, while the vertical header displays the row names, if any. `QHeaderView` is the class responsible for displaying headers. It has a number of parameters, such as whether the column can be moved (`setMovable`) and the `defaultAlignment` of the labels, which,

as we will see later, can be overridden by the model for specific columns. By default, the labels are centered. Here, we specify left alignment for the column labels:

```
header <- view$horizontalHeader()
header$defaultAlignment <- Qt$Qt$AlignLeft
```

Aesthetic properties `QTableView` provides a number of aesthetic features. By default, a grid is drawn that delineates the cells. We can set `showGrid` to "FALSE" to disable this. If a table has more than a few columns, it may be a good idea to fill the row backgrounds with alternating colors:

```
view$alternatingRowColors <- TRUE
```

Memory management

A view keeps a reference to its model, and the model method returns the model object. However, we offer a word of caution: since multiple views can refer to a single model, a view does not own its model. This means that if a model becomes inaccessible to R, i.e., if it goes out of scope, the model will be garbage collected, due to lack of an owner. For example, this does not work:

```
broken_view <- Qt$QTableView()
broken_view$setModel(qdataFrameModel(mtcars))
gc()
```

```
broken_view$model() # NULL, garbage collected
```

NULL

To prevent this, we should either (1) maintain a reference to the model in R, which we typically do in this text, or (2) explicitly give the view ownership of the model by setting the view as the parent of the model, like this:

```
parental_view <- Qt$QTableView()
broken_view$setModel(qdataFrameModel(mtcars,
                                     parent = parental_view))
gc()
```

```
broken_view$model() # not garbage collected
```

DataFrameModel instance

Formatting cells

Let us now assume that a missing value (NA) has been introduced into our data set:

```
qdataFrame(model)$mpg[1] <- NA
```

The table view will display this as "nan" or "inf", which is inconsistent with the notation of R. The conversion of the numeric data to text is carried out by an *item delegate*. Similar to a GTK+ cell renderer (Section 9.1), an item delegate is responsible for the rendering and editing of items (cells) in a view. Every type of item delegate is derived from the `QAbstractItemDelegate` class. By default, views in Qt will use an instance of `QStyledItemDelegate`, which renders items according to the current style. As Qt is unaware of the notion and encoding of missing values in R, we need to give Qt extra guidance. The `qtbase` package provides the `RTextFormattingDelegate` class for this purpose. To use it, we create an instance and set it as the item delegate for the view:

```
delegate <- qrTextFormattingDelegate()
view$setItemDelegate(delegate)
```

Delegates can also be assigned on a per-column or per-row basis. `RTextFormattingDelegate` will handle missing values in numeric vectors, as well as adhere to the numeric formatting settings in `options()`, namely "digits" and "scipen".

Column sizing

Managing the column widths of a table view is a challenge. This section will describe some of the strategies and suggest some best practices. The appropriate strategy depends, in part, on whether the table is expanding in its container.

When the table view is expanding, it will not necessarily fill its available space. To demonstrate,

```
model <- qdataFrameModel(mtcars[,1:5])
view <- Qt$QTableView()
view$setModel(model)
window <- Qt$QWidget()
window$resize(1000, 500)
vbox <- Qt$QVBoxLayout()
vbox$addWidget(view)
window$setLayout(vbox)
```

There is a gap between the last column and the right side of the window. It is difficult to appropriately size the columns of an expanding table. The simplest solution is to expand the last column:

```
header <- view$horizontalHeader()
header$stretchLastSection <- TRUE
```

To avoid the last column being too large, we can set pixel widths on the other columns. The simplest approach is to set the `defaultSectionSize` property, which gives all of the columns (except for the last) the same initial size:

```
header$defaultSectionSize <- 150
header$stretchLastSection <- TRUE
```

This usually yields an appropriate initial sizing. To resize specific columns, we could call `resizeSection`. Although specifying exact pixel sizes is inherently inflexible, the user is still free to adjust the column widths.

If, instead, we wish to pack a table so that it is not expanding, it may be desirable to initialize the column widths so that the columns optimally fit their contents:

```
view$resizeColumnsToContents()
```

This will need to be called each time the contents change.

By default, the size is always under control of the user (and the programmer), although this depends on the resize mode. The `resizeMode` property represents the default resize mode for all columns, and it defaults to "Interactive". The other modes are "Fixed", "Stretch" (expanding), and "ResizeToContents" (constrained to width needed to fit contents). The `setResizeMode` method changes the resize mode of a specific column. Below, we make all of our columns expand:

```
header$resizeMode(Qt$QHeaderView$Stretch)
```

The drawback to any of these modes is that the resizing is no longer interactive: the user cannot tweak the column widths.

When the size of a column is reduced such that it can no longer naturally display its contents, special logic is necessary. By default, `QTableView` will wrap text at word boundaries. This is controlled by the `wordWrap` property. When a single word is too long, the text will be ellipsized, i.e., truncated and appended with "...". This can be disabled with

```
view$textElideMode <- Qt$Qt$ElideNone
```

When the user attempts to reduce the size of a column to the point where ellipsizing would be necessary, it may be preferable to reduce instead the widths of the other columns. This mode is enabled with

```
header$cascadingSectionResizes <- TRUE
```

15.2 Displaying lists

It is often desirable to display a list of items, usually as text. A single column `QTableView` approximates this but also includes row and column headers, by default. Also, the two-dimensional API of `QTableView` is more complicated than needed for a one-dimensional list. For these and other reasons, Qt provides `QListView` for displaying a single column from a `QAbstractItemModel` as a list. We can use `DataFrameModel` to display the first column from a data frame (or anything coercible into a data frame) quickly:

```
model <- qdataFrameModel(rownames(mtcars))
view <- Qt$QListView()
view$setModel(model)
```

By default, `QListView` displays the first column from the model, although the column index can be customized.

Using a data model allows us to share data between multiple views. For example, we could view a data frame as a table using a `QTableView` and also display the row identifiers in a separate list:

```
mtcars_id <- cbind(makeAndModel = rownames(mtcars), mtcars)
model <- qdataFrameModel(mtcars_id)
table_view <- Qt$QTableView()
table_view$setModel(model)
##
list_view <- Qt$QListView()
list_view$setModel(model)
```

Now, when we resort the model, both views will be updated:

```
DF <- qdataFrame(model)
qdataFrame(model) <- DF[order(DF$mpg),]
```

The `QStringListModel` class When the list items are not associated with a data frame, they may be conveniently represented as a character vector. In this case, `DataFrameModel` is not very appropriate, as the character vector will be coerced to a data frame. Instead, consider `QStringListModel` from Qt. In `qtbase`, `QStringList` refers to a character vector. We demonstrate the use of `QStringListModel` to populate a list view from a character vector:

```
model <- Qt$QStringListModel(rownames(mtcars))
list_view <- Qt$QListView()
list_view$setModel(model)
```

Now we can retrieve the values as a character vector with the `stringList` method, rather than as a data frame:

```
head(model$stringList())
```

```
[1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
[4] "Hornet 4 Drive"     "Hornet Sportabout"  "Valiant"
```

`QListView` supports features beyond those of a simple list, including features often found in file browsers and desktops. For example, the items can be wrapped into additional columns or displayed in an icon mode. The widget also supports unrestricted layout and drag-and-drop.

15.3 Model-based combo boxes

Combo boxes were previously introduced as containers of string items and accompanying icons. The high-level API is sufficient for most uses; however, it is beneficial to understand that a combo box displays its pop-up menu with a `QListView`, which is based on a `QStandardItemModel` by default. It is possible to provide a custom data model for the list view. Explicitly leveraging the MVC pattern with a combo box affords greater aesthetic control and facilitates synchronizing the items with other views.

For example, we can create a combo box that lists the same cars that are present in our table and list views:

```
combo_box <- Qt$QComboBox()
combo_box$setModel(model)
```

By default, the first column from the model is displayed; this is controlled by the `modelColumn` property.

15.4 Accessing item models

We have shown how `DataFrameModel` and `QStringListModel` allow the storage and retrieval of data in familiar data structures. However, this is not true of all data models, including most of those in Qt. Alternative models are required, for example, in the case of hierarchical data. In such cases, or when interpreting user input, such as selection, it is necessary to interact with the low-level, generic API of the item/view framework.

An item model refers to its rows, columns, and cells with `QModelIndex` objects, which are created by the model:

```
index <- model$index(0, 0)
c(row = index$row(), column = index$column())
```

```
row column
0      0
```

Our "index" refers to the first row of the `QStringListModel`, using 0-based indices. The index points to a cell in the model, and we can retrieve the data in the cell using only the index:

```
(first_car <- index$data())
```

```
[1] "Mazda RX4"
```

We vectorize the above to retrieve all of the items in the list:

```
items <- sapply(seq(model$rowCount()), function(i) {  
  model$index(i - 1, 0)$data()  
})  
head(items, n=6)
```

```
[1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"  
[4] "Hornet 4 Drive"     "Hornet Sportabout"  "Valiant"
```

Setting the data is also possible, yet it requires calling `setData` on the model, not the index:

```
model$setData(index, toupper(first_car))
```

```
[1] TRUE
```

We will leave the population of a model with the low-level API as an exercise for the reader. Recall that `DataFrameModel` and `QStringListModel` provide an interface that is much faster and more convenient. When using such models, it is usually necessary to directly manipulate a `QModelIndex` only when handling user input, as we describe in the next section.

15.5 Item selection

Selection is likely the most common type of user interaction with lists and tables. There are five selection modes for item views; they are defined by the `QAbstractItemView::SelectionMode` enumeration and include:

- "SingleSelection" mode: allows only a single item to be selected at once.
- "ExtendedSelection" mode: the default; supports canonical multiple selection, where a range of items is selected by clicking the end points while holding the Shift key; clicking with the Ctrl key pressed adds arbitrary items to the selection.
- "ContiguousSelection" mode: disallows the Ctrl key behavior.
- "MultiSelection" mode: allows selection on mouse-over, with range selection by clicking and dragging.

We configure our list view for single selection with:

```
list_view$selectionMode <- Qt$QAbstractItemView$SingleSelection
```

For our tabular view, selection may be row-wise, column-wise, or item-wise (GTK+, by comparison, supports only row-wise selection). By default, selection is by item. While this is common in spreadsheets, we usually desire row-wise selection in a table, so we will override the default:

```
table_view$selectionBehavior <- Qt$QAbstractItemView$SelectRows
```

Accessing the selection

The selection state is stored in its own data model, `QItemSelectionModel`:

```
selection_model <- list_view$selectionModel()
```

This design allows views to synchronize selection. It also supports views on the selection state, such as a label indicating how many items are selected, independent of the particular type of item view.

We can query the selection model for the selected items in our list. Let us assume that we have selected the third row. We retrieve the data (label) in that row:

```
indices <- selection_model$selectedIndexes()
indices[[1]]$data()
```

```
[1] "Datsun 710"
```

When multiple selection is allowed, we must take care to interpret the selection efficiently, especially if a table has many rows. In the above, we obtained the selected indices. A selection is more formally represented by a `QItemSelection` object, which is a list of `QItemSelectionRange` objects. Under the assumption that the user has selected three separate ranges of items from the list view, we retrieve that selection from the selection model:

```
selection <- selection_model$selection()
```

Next, we coerce the `QItemSelection` to an explicit list of `QItemSelectionRange` objects and generate a vector of the selected indices:

```
indicesForSelection <- function(selection) {
  selection_ranges <- as.list(selection)
  unlist(lapply(selection_ranges, function(range) {
    seq(range$top(), range$bottom())
  }))
}
indicesForSelection(selection)
```

```
[1] 3 4 5 10 11 12 13 14 15 16 20 21 22 23 24
```

Coercion with `as.list` is possible for any class extending `QList`; `QItemSelection` is the only such class the reader is likely to encounter. Usually, the user selects a relatively small number of ranges, although the ranges can be wide. Looping over the ranges, but not the individual indices, will be significantly more efficient for large selections.

Responding to selection changes

To respond to a change in selection, connect to the `selectionChanged` signal on the selection model:

```
selected_indices <- rep(FALSE, nrow(mtcars))
selectionChangedHandler <- function(selected, deselected) {
  selected_indices[indicesForSelection(selected)] <<- TRUE
  selected_indices[indicesForSelection(deselected)] <<- FALSE
}
qconnect(selection_model, "selectionChanged",
         selectionChangedHandler)
```

The change in selection is communicated as two `QItemSelection` objects: one for the selected items, the other for the deselected items. We update a vector of the selected indices according to the change.

Assigning the selection

It is also possible to change the selection programmatically. For example, we may wish to select the first list item:

```
list_view$setCurrentIndex(model$index(0, 0))
```

This approach is simple but only supports selecting a single item. The selection is most generally modified by calling the `select` method on the selection model:

```
selection_model$select(model$index(0, 0),
                      Qt::QItemSelectionModel::Select)
```

The second argument describes how the selection is to be changed with regard to the index. It is a flag value and thus can specify several options at once, all listed in `QItemSelectionModel::SelectionFlags`. In the above, we issued the "Select" command. Other commands include "Deselect" and "Toggle". Thus, we could deselect the item in similar fashion:

```
selection_model$select(model$index(0, 0),
                      Qt::QItemSelectionModel::Deselect)
```

To select a range of items, efficiently we construct a `QItemSelection` object and set it on the model:

```
selection <- Qt$QItemSelection(model$index(3, 0),
                              model$index(10, 0))
selection_model$select(selection,
                      Qt$QItemSelectionModel$Select)
```

We have selected items 4 to 11. Multiple ranges can be added to the `QItemSelection` object by repeatedly calling its `select` method.

Querying a selection in a table view is essentially the same as for the list view, except we can request indices representing entire rows or columns. In this example, we are interested in the rows, where the user has selected:

```
selection_model <- table_view$selectionModel()
sapply(selection_model$selectedRows(), qinvoke, "row")
```

```
list()
```

We invoke the `row` method on each returned `QModelIndex` object to get the row indices.

When we are setting the selection, there are conveniences for selecting an entire row or column. We select the first row of the table:

```
table_view$selectRow(0)
```

Selecting a range of rows is very similar to selecting a range of list items, except we need to add the "Rows" selection flag:

```
selection_model$select(selection,
                      Qt$QItemSelectionModel$Select | Qt$QItemSelectionModel$Rows)
```

15.6 Sorting and filtering

One of the benefits of the MVC design is that models can serve as proxies for other models. Two common applications of proxy models are sorting and filtering. Decoupling the sorting and filtering from the source model avoids modifying the original data. The filtering and sorting is dynamic, in the sense that no data is actually stored in the proxy. The proxy delegates to the child model, while mapping indices between the filtered and unfiltered (or sorted and unsorted) coordinate space. Thus, there is little cost in memory.

Qt implements both sorting and filtering in a single class: `QSortFilterProxyModel`.¹ After constructing an instance and specifying the child model, the proxy model can be handed to a view like any other model:

```
proxy_model <- Qt$QSortFilterProxyModel()
proxy_model$setSourceModel(model)
table_view$setModel(proxy_model)
list_view$setModel(proxy_model)
```

¹In `Qt2` there are separate proxy models for sorting and filtering, cf. Section 9.1.

Our views will now draw data through the proxy, rather than from the original model.

Both table views and treeviews provide interfaces for the user to sort the underlying model. The user clicks on a column header to sort by the corresponding column. Clicking multiple times toggles the sort order. This behavior is enabled by setting the `sortingEnabled` property:

```
table_view$sortingEnabled <- TRUE
```

Since the sort occurs in the model, both the table view and list view display the sorted data. The sort has been applied to both the table and list view. It is also possible to sort programmatically by calling the `sort` method, passing the index of the sort column. We sort our data by the "mpg" variable:

```
proxy_model$sort(1) # mpg in column 2 of model
```

The built-in sorting logic understands basic data types such as strings and numbers. Customizing the sorting requires overriding the `lessThan` virtual method in a new class.

`QSortFilterProxyModel` supports filtering by row. The column indicated by the `filterKeyColumn` property is matched against a string pattern. Only rows with a matching value in the key column are allowed past the filter. The pattern is a `QRegExp`, which supports several different syntax forms, including: fixed strings, wild cards (globs), and regular expressions. For example, we can filter for cars made by Mercedes:

```
proxy_model$filterKeyColumn <- 0
proxy_model$filterRegExp <- Qt$QRegExp("^Merc")
```

This approach should satisfy the majority of use cases. To achieve more complex filtering, including filtering of columns, subclassing is necessary.

It is also possible to hide rows and columns at the view by calling `setColumnHidden` or `setRowHidden`. For example, we hide the "Price" column (column 5):

```
table_view$setColumnHidden(5 - 1L, TRUE)
```

It is common for different views to display different types of information, which translates to different sets of columns. For row filtering, the proxy model approach is usually preferable to hiding view rows, as the filtering will apply to all views of the data.

15.7 Decorating items

Thus far, we have considered only the display of plain text in item views. To move beyond this, the model needs to communicate extra rendering information to the view. With GTK+, this information is stored in extra

columns, which are mapped to visual properties. Unlike GTK+, however, Qt does not require every cell in a column to have the same rendering strategy or even the same type of data. Thus, Qt stores rendering information at the item level. An item is actually a collection of data elements, each with a unique *role* identifier. The mapping of roles to visual properties depends on the `QItemDelegate` associated with the item. The default item delegate, `QStyledItemDelegate`, understands most of the standard roles listed in the `Qt::ItemDataRole` enumeration, selectively listed in Table 15.7.

For example, when we create a `DataFrameModel`, the default behavior is to associate the data-frame values with the `Qt$DisplayRole`. `QStyledItemDelegate` (and its extension `RTextFormattingDelegate`) convert the value to a string for display. Other roles control aspects like the background and foreground colors, the font, and the decorative icon, if any.

DataFrameModel roles `DataFrameModel` instances support role-specific values for each item, provided `"useRoles = TRUE"` is passed to the constructor. It is then up to the programmer to indicate the mapping from a data-frame column to a column and role in the model. The mapping is encoded in the column names. Each column name should have the syntax `"[.NAME][.ROLE]"`, where `"NAME"` indicates the column name in the model² and `"ROLE"` refers to a value in `Qt::ItemDataRole`, without the `"Role"` suffix. If the column name does not contain a period (i.e., there is no `"ROLE"`), the display role is assumed.

For example, to customize the row names, we could shade the background of the first column, the makes and models, in gray:

```
mtcars_id <- cbind(makeAndModel=rownames(mtcars), mtcars)
model <- qdataFrameModel(mtcars_id, useRoles = TRUE)
qdataFrame(model)$ .makeAndModel.background <-
  list(qcolor("gray"))
```

In the above, we store a list of `QColor` instances in our data frame.³

The set of supported data types for each role depends on the delegate. For delegates derived from `QStyledItemDelegate`, see the documentation for that class. Due to implicit conversion in the internals of Qt, the number of possible inputs is much greater than those explicitly documented. For example, the `"background"` role demonstrated above formally accepts a `QBrush` object, while implicit conversion allows types such as `QColor` and `QGradient`.

²`"NAME"` can refer to multiple columns, if separated by periods, or all columns if omitted.

³Storing objects other than atomic vectors in a data frame requires some care, which we avoid here. If we had added that column in a call to `"data.frame"` or `cbind`, it would have been necessary to wrap the list with `I()` in order to prevent coercion of the list to a data frame.

makeAndModel	mpg	cyl	disp	hp	drat
Mazda RX4	21	6	160	110	3.9
Mazda RX4 ...	21	6	160	110	3.9
Datsun 710	22.8	4	108	93	3.85
Hornet 4 Dr...	21.4	6	258	110	3.08
Hornet Spor...	18.7	8	360	175	3.15

Figure 15.2: Example decorating cell items using role specification of `DataFrameModel`.

It is possible for a single data frame column to specify the values for a particular role across multiple model columns. This is useful, for example, when modifying the font uniformly across several columns of interest. Here, we bold the "mpg" and "hp" columns:

```
qdataFrame(model)$mpg.hp.font <-
  list(qfont(weight = Qt$QFont$Bold))
```

After these modifications, the model can be passed to a view, as in Figure 15.2.

```
view <- Qt$QTableView()
view$setModel(model)
view$verticalHeader()$hide() # hide default row names
```

If the "NAME" component is omitted, the role will apply to all columns for which a role of the same type has not already been specified. Here, we change the foreground color for all cells:

```
qdataFrame(model)$foreground <- list(qcolor("darkgray"))
```

Roles in other models For models other than `DataFrameModel`, we set data for a specific role by passing the optional role argument to the model's `setData` method. The value of role defaults to "EditRole", meaning that the data is in an editable form, i.e., it is treated as input from the user.

Here, we show how to create a list view and set the background of the first item to yellow:

```
list_model <- Qt$QStringListModel(rownames(mtcars))
list_model$setData(list_model$index(0, 0), "yellow",
  Qt$Qt$BackgroundRole)
list_view <- Qt$QListView()
list_view$setModel(list_model)
```

Table 15.1: 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)

15.8 Displaying hierarchical data

Hierarchical data is generally stored in `QStandardItemModel`, the primary implementation of `QAbstractItemModel` built into Qt. Hierarchical data in R often arises when splitting a tabular data set by some combination of factors. For our demonstration, we will display in a tree the result of splitting the `Cars93` data set by manufacturer. The first step of our demonstration is to create the model, with a single column:

```
tree_model <- Qt$QStandardItemModel(rows = 0, columns = 1)
```

We need to create an item for each manufacturer and store the corresponding records as its children:

```
by(Cars93, Cars93$Manufacturer, function(DF) {
  tree_model$insertRow(tree_model$rowCount())
  manufacturer <- tree_model$index(tree_model$rowCount()-1L, 0)
  tree_model$setData(manufacturer, DF$Manufacturer[1])
  tree_model$insertRows(0, nrow(DF), manufacturer)
  tree_model$insertColumn(0, manufacturer)
  for (i in seq_along(DF$Model)) {
    record <- tree_model$index(i-1L, 0, manufacturer)
    tree_model$setData(record, DF$Model[i])
  }
})
```

As before, we create a `QModelIndex` object for accessing each cell of the model (in line 3). We need to add rows and columns to each manufacturer node before creating its children (lines 5 and 6). This nested loop approach to populating a model is much less efficient than converting a

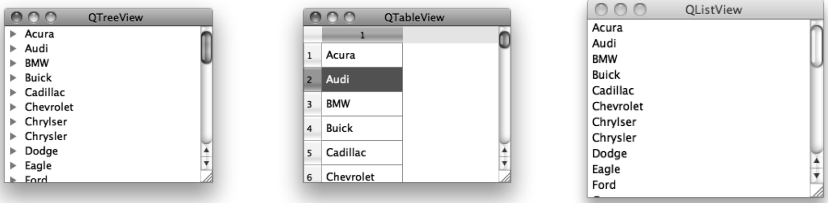


Figure 15.3: The `tree_model` instance viewed in a treeview, a table view and a list view.

`data.frame` to a `DataFrameModel`, but here it is necessary to communicate the hierarchical information.

The `QStandardItem` class In addition to implementing the `QAbstractItemModel` interface, `QStandardItemModel` also represents an item as a `QStandardItem` object. Many operations, including inserting, removing, and manipulating children, can be performed on a `QStandardItem`, instead of directly on the model. This may be convenient in some circumstances. For example, the code listed above for populating the model simplifies to:

```
by(Cars93, Cars93$Manufacturer, function(DF) {
  manufacturer <- as.character(DF$Manufacturer[1])
  manufacturer_item <- Qt$QStandardItem(manufacturer)
  tree_model$appendRow(manufacturer_item)
  children <- lapply(as.character(DF$Model), Qt$QStandardItem)
  lapply(children, manufacturer_item$appendRow)
})
```

The `QTreeView` widget displays the data in a table, with the conventional buttons on the left for expanding and collapsing nodes. We create an instance and set the model:

```
tree_view <- Qt$QTreeView()
tree_view$setModel(tree_model)
```

Columns in a `QStandardItemModel` can be named by calling `setHorizontalHeaderNames`, as shown in the workspace browser example, below. Often, as in our case, a treeview has only a single column. It may be desirable to hide that column header with

```
tree_view$headerHidden <- TRUE
```

Figure 15.3 shows the `tree_model` in the three separate types of views we've discussed, the leftmost being with a `QTreeView` instance, as just illustrated.

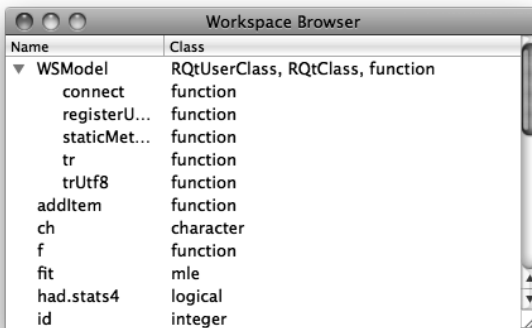


Figure 15.4: The completed workspace browser showing a hierarchical view of the objects in the global environment.

Example 15.1: A workspace browser

This example shows how to use the tree-widget item to display a snapshot of the current workspace. Figure 15.4 shows an illustration. Each object in the workspace maps to an item, where recursive objects with names will have their components represented in a hierarchical manner. In Example 12.1 we created a class `WSWatcher` to monitor the workspace for changes. Now we build on that example.

The following `addItem` function creates an item from a named component of a parent object and adds the new item under the given parent index:

```
addItem <- function(varname, parent_object, parent_item) {

  obj <- parent_object[[varname]]
  ## main interaction with tree model
  item <- Qt$QStandardItem(varname)
  class_item <- Qt$QStandardItem(paste(class(obj),
                                       collapse = ", "))
  parent_item$appendRow(list(item, class_item))

  ## Recursively create ancestor items, if needed
  nms <- NULL
  if (is.recursive(obj)) {
    if (is.environment(obj))
      nms <- ls(obj)
    else if (!is.null(names(obj)))
      nms <- names(obj)
  }
}
```

```
sapply(nms, addItem, parent_item = item,
       parent_object = obj)
}
```

Our main function is one called when changes are made to the workspace. There are two cases: when we need to remove expired items and when we need to add new ones.

```
updateTopLevelItems <- function(ws_watcher, view,
                               env = .GlobalEnv) {
  ## remove these (by index)
  remove <- ws_watcher$changedVariables()
  cur_shown <- sapply(seq(model$rowCount()),
                     function(i) model$index(i - 1, 0)$data())
  indices_to_remove <- which(cur_shown == remove)
  indices_to_remove <- sort(inds_to_remove, decreasing=TRUE)
  ## add these (by variable name)
  new_names <- ws_watcher$addedVariables()

  ## replace/add these
  model <- view$model()
  view$updatesEnabled <- FALSE
  if(length(indices_to_remove))
    sapply(indices_to_remove - 1L, model$removeRow)
  ## add
  sapply(new_names, addItem, parent_object = env,
         parent_item = model$invisibleRootItem())
  model$sort(0, Qt$Qt$AscendingOrder)
  view$updatesEnabled <- TRUE
}
```

We remove objects corresponding to expired digests by their index. We need to sort the indices in decreasing order so as not to invalidate any indices along the way. Then we add in new or changed variable names. Finally, the model is sorted. We set the `updatesEnabled` property to freeze the view while the model is updated to make a smoother transition.

This function is used to initialize the view:

```
initializeTopLevelItems <- function(ws_watcher, view,
                                   env = .GlobalEnv)
{
  current_names <- ws_watcher$objects
  model <- view$model()
  view$updatesEnabled <- FALSE
  sapply(current_names, addItem, parent_object = env, # add
         parent_item = model$invisibleRootItem())
  model$sort(0, Qt$Qt$AscendingOrder)
  view$updatesEnabled <- TRUE
}
```

Finally, we construct the model and view:

```
model <- Qt$QStandardItemModel(rows = 0, columns = 2)
model$setHorizontalHeaderLabels(c("Name", "Class"))
view <- Qt$QTreeView()
view$windowTitle <- "Workspace Browser"
view$headerHidden <- FALSE
view$setModel(model)
```

This last call initializes the workspace model and display:

```
ws_watcher <- WSWatcher()
ws_watcher$updateVariables()
initializeTopLevelItems(ws_watcher, view)
```

Assuming we are updating the workspace model by some means, all that remains is calling the function to update the top-level items as needed:

```
qconnect(ws_watcher, "objectsChanged", function()
  updateTopLevelItems(ws_watcher, view))
```

15.9 User editing of data models

Some data models, including `DataFrameModel`, `QStringListModel`, and `QStandardItemModel`, support modification of their data. To determine whether an item may be edited, call the `flags` method on the model, passing the index of the item, and check for the `ItemIsEditable` flag:

```
(tree_model$index(0, 0)$flags() & Qt$Qt$ItemIsEditable) > 0
```

```
[1] TRUE
```

To enable editing on a column in a `DataFrameModel`, it is necessary to specify the edit role for the column. For example, we might add a logical column named `Analyze` to the `mtcars` data frame for indicating whether a record should be included in an analysis. In the view, the user will be able to use a combo box to choose between `TRUE` and `FALSE`. We could display an editable `Analyze` column by adding a column named `.Analyze.edit`, but instead we take advantage of a convenience of `DataFrameModel`. We simply add the `Analyze` column and pass its name as the editable argument to `qdataFrameModel`:

```
DF <- mtcars
DF$Analyze <- TRUE
model <- qdataFrameModel(DF, editable = "Analyze")
```

If a view is assigned an editable model, it will enter its editing mode upon a certain trigger. By default, derivatives of `QAbstractItemView` will

initiate editing of an editable column upon double mouse-button click or a key press. This is controlled by the `editTriggers` property, which accepts a combination of `QAbstractItemView::EditTrigger` flags. For example, we could disable editing through a view:

```
view$editTriggers <- Qt$QAbstractItemView$NoEditTriggers
```

When editing is requested, the view will pass the request to the delegate for the item. The standard item delegate, `QStyledItemDelegate`, will present an editing widget created by its instance of `QItemEditorFactory`. The default item editor factory will create a combo box for logical data, a spin box for numeric data, and a text-edit box for character data. Other types of data, like times and dates, are also supported. To specify a custom editor widget for some data type, it is necessary to subclass `QItemEditorCreatorBase` and register an instance with the item editor factory.

15.10 Drag-and-drop in item views

The item views have native support for drag-and-drop. All of the built-in models, as well as `DataFrameModel`, communicate data in a common format so that drag-and-drop works automatically between views. `DataFrameModel` also provides its data in the R serialization format, corresponding to the "application/x-rlang-transport" MIME type. This facilitates implementing custom drop targets for items in R.

Dragging is enabled by setting the `dragEnabled` property to "TRUE":

```
view$dragEnabled <- TRUE
```

Enabling drops is the same as for any other widget, with one addition:

```
view$acceptDrops <- TRUE
view$showDropIndicator <- TRUE
```

The second line tells the view to indicate visually where the item will be dropped.

The following enables moving items within a view, i.e., reordering:

```
view$dragDropMode <- Qt$QAbstractItemView$InternalMove
```

However, that will prevent receiving drops from other views, and dragging to other views will always be a move, not a copy.

Although we have enabled drag-and-drop on the view, the level of support actually depends on the model. The supported actions can be queried with `supportedDragActions` and `supportedDropActions`. The item flags determine whether an individual item can be dragged or dropped upon. Most of the built-in models will support both copy and move actions, when dragging or dropping. `DataFrameModel` supports only copy actions when dragging; dropping is not supported.

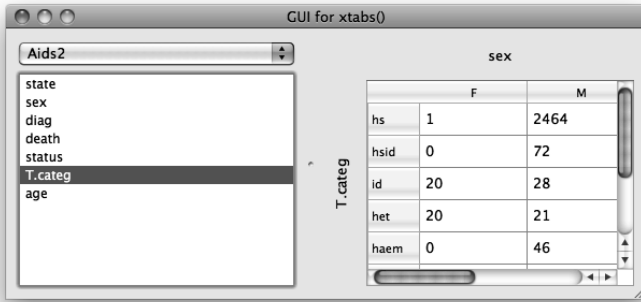


Figure 15.5: A table widget to display contingency tables and a means to specify the variables through drag and drop.

Example 15.2: A drag-and-drop interface to xtabs

This example uses a table view to display the output from `xtabs`. To specify the variables, the user drags variable names from a list to one of two labels, representing terms in the formula.

A VariableSelector class First, we define the `VariableSelector` widget, which contains a combo box for choosing a data frame and a list view for the variable names. When a data frame is chosen in the combo box, its variables are shown in the list:

```
qsetClass("VariableSelector", Qt$QWidget,
          function(parent = NULL) {
    super(parent)
    ## widgets
    this$df_combo_box <- Qt$QComboBox()
    this$variable_list <- Qt$QListView()
    this$variable_list$setModel(
      qdataFrameModel(data.frame(), this,
                      useRoles = TRUE))
    this$variable_list$dragEnabled <- TRUE

    ## layout
    layout <- Qt$QVBoxLayout()
    layout$addWidget(df_combo_box)
    layout$addWidget(variable_list)
    variable_list$setSizePolicy(Qt$QSizePolicy$Expanding,
                               Qt$QSizePolicy$Expanding)

    setLayout(layout)

    updateDataSets()
  })
```

```
qconnect(df_combo_box, "activated(int)", function(ind) {
    this$dataFrame <- df_combo_box$currentText
})
})
```

This utility populates the combo box with a list of data frames, keeping the selected data frame if still valid.

```
qsetMethod("updateDataSets", VariableSelector, function() {
    current_text <- df_combo_box$currentText
    df_combo_box$clear()
    DFs <- ProgGUIinR:::avail_dfs(.GlobalEnv)
    if(length(DFs)) {
        this$df_combo_box$addItem(DFs)
        if(is.null(current_text) || !current_text %in% DFs) {
            this$df_combo_box$currentIndex <- -1
            this$dataFrame <- NULL
        } else {
            this$df_combo_box$currentIndex <-
                which(current_text == DFs)
            this$dataFrame <- current_text
        }
    }
})
```

The data frame is stored in the following call to `qsetProperty`. We overwrite the underlying write method to also update our model for the variable list, as well.

```
qsetProperty("dataFrame", VariableSelector,
    write = function(DF) {
        if (is.null(DF))
            DF <- data.frame()
        else if (is.character(DF))
            DF <- get(DF, .GlobalEnv)
        ##
        model <- variable_list$model()
        icons <- lapply(DF, getIcon)
        qdataFrame(model) <-
            data.frame(variable=names(DF),
                variable.decoration=I(icons))
        this$.dataFrame <- DF
        dataFrameChanged()
    })
```

When the property is written, the variable selector will emit this signal:

```
qsetSignal("dataFrameChanged", VariableSelector)
```

A `QLabel` subclass Next, a derivative of `QLabel` is defined that accepts drops from the variable list and is capable of rotating text for displaying the y -label component:

```
qsetClass("VariableLabel", Qt$QLabel, function(parent=NULL) {
  super(parent)
  this$rotation <- 0L
  setAcceptDrops(TRUE)
  setAlignment(Qt$Qt$AlignHCenter | Qt$Qt$AlignVCenter)
})
```

We define two properties, one for the rotation and the other for the variable name, which is not always the same as the label text:

```
qsetProperty("rotation", VariableLabel)
qsetProperty("variable_name", VariableLabel)
```

To enable client code to respond to a drop, we define a signal:

```
qsetSignal("variableNameDropped", VariableLabel)
```

This utility tries to extract a variable name from the MIME data, which `DataFrameModel` should have serialized appropriately:

```
variableNameFromMimeData <- function(mime_data) {
  name <- NULL
  RDA_MIME_TYPE <- "application/x-rlang-transport"
  if(mime_data$hasFormat(RDA_MIME_TYPE)) {
    name_list <- unserialize(mime_data$data(RDA_MIME_TYPE))
    if (length(name_list) && is.character(name_list[[1]]))
      name <- name_list[[1]]
  }
  name
}
```

To handle the drag events, we override the methods `dragEnterEvent`, `dragLeaveEvent`, and `dropEvent`. The first two simply change the background of the label to indicate a valid drop:

```
qsetMethod("dragEnterEvent", VariableLabel, function(event) {
  mime_data <- event$mimeType()
  if(!is.null(variableNameFromMimeData(mime_data))) {
    setForegroundRole(Qt$QPalette$Dark)
    event$acceptProposedAction()
  }
})
qsetMethod("dragLeaveEvent", VariableLabel, function(event) {
  setForegroundRole(Qt$QPalette$WindowText)
  event$accept()
})
```

To respond to a drop event, we get the variable name, set the text of the label, and emit the `variableNameDroppedVariableLabel` signal:

```
qsetMethod("dropEvent", VariableLabel, function(event) {
  setForegroundRole(Qt$QPalette$WindowText)
  mime_data <- event$mimeTypeData()
  this$variable_name <- variableNameFromMimeTypeData(mime_data)
  if(!is.null(variable_name)) {
    this$text <- variable_name
    variableNameDropped()
    setBackgroundRole(Qt$QPalette$Window)
    event$acceptProposedAction()
  }
})
```

To complete the `VariableLabel` class, we override the `paintEvent` event to respect the rotation property. Drawing low-level graphics is beyond our scope. In short, we translate the origin to the center of the label rectangle, rotate the coordinate system by the angle, then draw the text:

```
qsetMethod("paintEvent", VariableLabel, function(event) {
  painter <- Qt$QPainter()
  painter$begin(this)

  painter$save()
  painter$translate(width / 2, height / 2)
  painter$rotate(-(rotation))
  rect <- painter$boundingRect(0, 0, 0, 0,
                               Qt$Qt$AlignCenter, text)
  painter$drawText(rect, Qt$Qt$AlignCenter, text)
  painter$restore()
  painter$end()
})
```

An `XTabsWidget` class Our main widget consists of three child widgets: two drop labels for the formula 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:

```
qsetClass("XtabsWidget", Qt$QWidget, function(parent = NULL) {
  super(parent)
  initWidgets()
  initLayout()
})
```

We do not list the `initLayout` method, as it simply adds the widgets to a grid layout. The `initWidgets` method initializes three widgets:

```

qsetMethod("initWidgets", XtabsWidget, function() {
  this$xlabel <- VariableLabel()
  qconnect(xlabel, "variableNameDropped", invokeXtabs)

  this$ylabel <- VariableLabel()
  pt <- ylabel$font$pointSize()
  ylabel$minimumWidth <- 2*pt; ylabel$maximumWidth <- 2*pt
  ylabel$rotation <- 90L
  qconnect(ylabel, "variableNameDropped", invokeXtabs)

  this$table_view <- Qt$QTableView()
  table_view$setModel(qdataFrameModel(data.frame(), this))
  clearLabels()
})

```

The `xlabel` is straightforward: we construct it, then connect to the drop signal. For the `ylabel` we also adjust the rotation and constrain the width based on the font size (otherwise the label width reflects the length of the dropped text). The `clearLabels` method (not shown) just initializes the labels.

This function builds the formula, invokes `xtabs`, and updates the table view; we hide the conditional call to `xtabs`.

```

qsetMethod("invokeXtabs", XtabsWidget, function() {
  if (is.null(dataFrame))
    return()

  x <- xlabel$variable_name
  y <- ylabel$variable_name

  if(!is.null(table <- call_xtabs(dataFrame, x, y)))
    updateTableView(table)
})

```

We define a method to update the table view:

```

qsetMethod("updateTableView", XtabsWidget, function(table) {
  model <- table_view$model()
  if (length(dim(table)) == 1)
    qdataFrame(model) <- data.frame(count = unclass(table))
  else qdataFrame(model) <- data.frame(unclass(table))
})

```

Finally, we define a property for the data frame held in the `XtabsWidget` class:

```

qsetProperty("dataFrame", XtabsWidget,
  write = function(dataFrame) {
    clearLabels()
    this$.dataFrame <- dataFrame
  }
)

```

```
    })
```

All that remains is to place the `VariableSelector` and `XtabsWidget` together in a split pane and then connect a handler that keeps the data sets synchronized:

```
w <- Qt$QSplitter()
w$setWindowTitle("GUI for xtabs()")
w$addWidget(vs <- VariableSelector())
w$addWidget(tw <- XtabsWidget())
w$setStretchFactor(1, 1)
qconnect(vs, "dataFrameChanged", function() {
  tw$dataFrame <- vs$dataFrame
})
w$show(); w$raise()
```

Figure 15.5 shows the result after the user has dragged two variables onto the labels.

15.11 Widgets with internal models

While separating the model from the view provides substantial flexibility, in practice it is often sufficient and slightly more convenient to manipulate a view with a built-in data model. Qt provides a set of view widgets with internal models:

`QListWidget` for simple lists of items,

`QTableWidget` for a flat table, and

`QTreeWidget` for a tree table.

In our experience, the convenience of these classes is not worth the loss in flexibility and other advantages of the model/view design pattern. `QTableWidget`, in particular, precludes the use of `DataFrameModel`, so `QTableWidget` is usually not nearly as convenient or performant as the model-based `QTableView`. Thus, we are inclined to omit a detailed description of these widgets. However, we will describe `QListWidget`, out of an acknowledgement that displaying a short, simple list of items is a common task in a GUI.

Displaying short, simple lists

`QListWidget` is an easy-to-use widget for displaying a set of items for selection (Figure 15.6). As with combo boxes, we can populate the items directly from a character vector through the `addItem`s method:

```
list_widget <- Qt$QListWidget()
list_widget$addItem(state.name)
```

This saves one line of code compared to populating a `QListView` via a `QStringListModel`. To clear a list of its items, call the `clear` method. Passing an item to `takeItem` will remove that specific item from the widget.

The items in a `QListWidget` instance are of the `QListWidgetItem` class. New items can be constructed directly through the constructor:

```
item <- Qt$QListWidgetItem("Puerto Rico", list_widget)
```

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 method `addItem` or the method `insertItem`, for inserting to a specific instance.

To retrieve an item given its index, we call the `item` method:

```
first <- list_widget$item(0)
first$text()
```

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

Many aspects of an item can be manipulated. These roughly correspond to the built-in roles of items in `QAbstractItemModel`. We can specify the text, font, icon, status and tooltips, as well as foreground and background colors.

By default, `QListWidget` allows only a single item to be selected simultaneously. As with other `QAbstractItemView` derivatives, this may be adjusted to allow multiple selection through the `selectionMode` property:

```
list_widget$selectionMode <- Qt$QListWidget$ExtendedSelection
```

We can programmatically select the states that begin with "A":

```
sapply(grep("^A", state.name),
       function(i) list_widget$item(i - 1)$setSelected(TRUE))
```

The method `selectedItems` will return the selected items in a list:

```
selected_items <- list_widget$selectedItems()
sapply(selected_items, qinvoke, "text")
```

```
[1] "Alabama" "Alaska" "Arizona" "Arkansas"
```

To handle changes in the selection, connect to `itemSelectionChanged`:

```
qconnect(list_widget, "itemSelectionChanged", function() {
  selected <- list_widget$selectedItems()
  selected_text <- sapply(selected, qinvoke, "text")
  message("Selected: ", paste(selected_text, collapse = ", "))
})
```

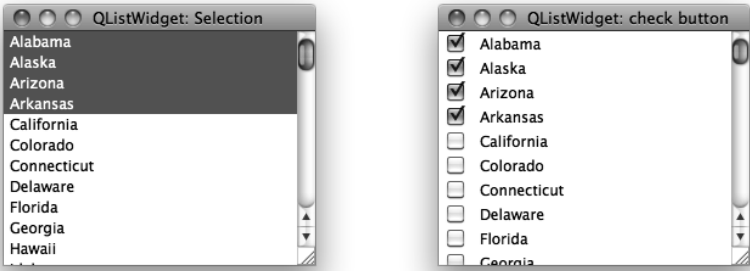



Figure 15.6: Two easily implemented styles for selecting items from a `QListWidget` instance: the traditional selection and using check buttons.

Using check buttons for selection It is often easier for the user to select multiple items by clicking check buttons next to the desired items. The right figure in Figure 15.6 shows an example. The check box is shown only if we explicitly set the check state of item. The possible values are "Checked", "Unchecked", and "PartiallyChecked". Here, we set all of the items to unchecked to show the check buttons, check the selected items, then turn off selection.

```
items <- sapply(seq(list_widget$count) - 1L, list_widget$item)
sapply(items, qinvoke, "setCheckState", Qt$Qt$Unchecked)
## check selected
selected <- list_widget$selectedItems()
sapply(selected, function(x) x$setCheckState(Qt$Qt$Checked))
## clear selection now
list_widget$selectionModel()$clear()
list_widget$selectionMode <- Qt$QListWidget$NoSelection
```

To get the selected items, we can iterate over the items, as above, and invoke the `checkedState` method:

```
state <- sapply(items, "qinvoke", "checkedState")
head(state, n = 8) # 2 is checked, 0 not
```

```
[1] 2 2 2 2 0 0 0 0
```

For long lists, this looping will be time consuming. In such cases, it is likely preferable to use `QListView`, `DataFrameModel`, and the "Checked-StateRole".

15.12 Implementing custom models

Normally, the `DataFrameModel` and the models in Qt are sufficient. We can imagine other cases, however. For example, we might need to view an instance of a formal reference class that conforms to a tabular or hierarchical structure. In such case, it may be appropriate to implement a custom model in R. We warn the reader that this is a significant undertaking, and, unfortunately, custom models do not scale well, due to frequent callbacks into R.

Required methods The basic interface of a model requires that at a minimum the methods `rowCount`, `columnCount`, and `data` be provided. The first two describe the size of the table for any views. We have already demonstrated the use of the `data` method in the previous sections. It provides data to the view for a particular cell *and* role. For example, if we are displaying numeric data, the `DisplayRole` might format the numeric values (showing a fixed number of digits, say), yet the `EditRole` role might display all the digits so accuracy is not lost. If a role is not implemented, a value of `NULL` should be returned. We can also implement the `headerData` method to populate the view headers.

Editable models For editable models, we must also implement the `flags` method to return a flag containing `ItemIsEditable` and the `setData` method. When a value is updated, we should call the `dataChanged` method to notify the views that a portion of the model is changed. This method takes two indices, which together specify a rectangle in the table.

To provide for resizable tables, Qt requires us to notify the views about dimension changes. For example, an implemented `insertColumns` should call `beginInsertColumns` before adding the column to the model and then `endInsertColumns` just after.

Example 15.3: Using a custom model to edit a data frame

This example shows how to create a custom model to edit a data frame. Given that `DataFrameModel` supports editing, there is actually no reason to use this model. The purpose is to illustrate the steps in model implementation. The performance is poor compared to that of `DataFrameModel`, 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, instead of overriding the `headerData` method, which the Qt views call far too often.

Our basic constructor simply assigns to a `dataframe` property the data frame passed to it.

```
qsetClass("DfModel", Qt$QAbstractTableModel,
         function(Df = data.frame(V1 = character(0)),
```

mpg	cyl	disp	hp	drat	wt
21.00	6	160.00	110.00	3.90	2.62
21.00	6.00	160.00	110.00	3.90	2.88
22.80	4.00	108.00	93.00	3.85	2.32
21.40	6.00	258.00	110.00	3.08	3.21
18.70	8.00	360.00	175.00	3.15	3.44

Figure 15.7: A view providing a means to edit a data frame's contents. The underlying model subclasses `QAbstractTableModel`, trading of the ability to customize for a lack of responsiveness.

```

        parent = NULL)
    {
        super(parent)
        this$DF <- DF
    })

```

Here, we configure the dataframe property, implementing a write method so that assigning to this property will call the `dataChanged` method to notify any views of a change:

```

qsetProperty("DF", DfModel, write = function(DF) {
    this$.DF <- DF
    dataChanged(index(0, 0), index(nrow(DF), ncol(DF)))
})

```

As mentioned, there are three virtual methods required by the interface: `rowCount`, `columnCount`, and `data`. The first two delegate down to `nrow` and `ncol`:

```

qsetMethod("rowCount", DfModel,
    function(index) nrow(this$DF) + 1)
qsetMethod("columnCount", DfModel,
    function(index) ncol(this$DF))

```

The `data` method is then the main method to implement. Here, we wish to customize the data display based on the class of the variable represented in a column, a natural use of S3 methods, which dispatch on exactly that. Here is a method for defining the display role:

```

display_role <- function(x, row, ...) UseMethod("display_role")
display_role.default <- function(x, row)
    sprintf("%s", x[row])

```

```
display_role.numeric <- function(x, row)
  sprintf("%.2f", x[row])
display_role.integer <- function(x, row)
  sprintf("%d", x[row])
```

We see that numeric values are formatted to have decimal points. The data is still stored in its native form; a string is returned only for display. An alternative approach would be to provide the raw data and rely on `RTextFormattingDelegate` to display the numeric values according to the current R configuration. However, the above approach generalizes basic numeric formatting.

Our data method has this basic structure (we avoid showing the cases for all the different roles):

```
qsetMethod("data", DfModel, function(index, role) {
  row <- index$row()
  col <- index$column() + 1

  if(role == Qt$Qt$DisplayRole) {
    if(row > 0)
      display_role(DF[,col], row)
    else
      names(DF)[col]
  } else if(role == Qt$Qt$EditRole) {
    if(row > 0)
      as.character(DF[row, col])
    else
      names(DF)[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$ItemIsEnabled)
  } else {
    current_flags <- super("flags", index)
    return(current_flags | Qt$Qt$ItemIsEditable)
  }
})
```

To edit cells we also need to implement a method to set the data once edited. Since the data method provides a string for the edit role, `setData`

will be passed one, as well. We define some methods on the S3 generic `fit_in`, which will coerce the string to the original type. For example:

```
fit_in <- function(x, value) UseMethod("fit_in")
fit_in.default <- function(x, value) value
fit_in.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:

```
qsetMethod("setData", DfModel, function(index, value, role) {
  if(index$isValid() && role == Qt$Qt$EditRole) {
    DF <- this$DF
    row <- index$row()
    col <- index$column() + 1

    if(row > 0) {
      x <- DF[, col]
      DF[row, col] <- fit_in(x, value)
    } else {
      names(DF)[col] <- value
    }
    this$DF <- DF
    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 us to replace a column of values:

```
qsetMethod("setColumn", DfModel, function(col, value) {
  ## pad with NA if needed
  n <- nrow(this$DF)
  if(length(value) < n)
    value <- c(value, rep(NA, n - length(value)))
  value <- value[1:n]
  DF <- this$DF
  DF[,col] <- value
  this$DF <- DF # 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) {
  DF <- this$DF
  if(name %in% names(DF)) {
    return(setColumn(min(which(name == names(DF))), value))
  }
  beginInsertColumns(Qt$QModelIndex(),
                    columnCount(), columnCount())
  DF[[name]] <- value
  this$DF <- DF
  endInsertColumns()
  return(TRUE)
})
```

To demonstrate our model, we construct an instance and set it on a view:

```
model <- DfModel(mtcars)
view <- Qt$QTableView()
view$setModel(model)
```

Finally, we customize the view by defining the edit triggers and hiding the row and column headers:

```
trigger_flag <- Qt$QAbstractItemView$DoubleClicked |
               Qt$QAbstractItemView$SelectedClicked |
               Qt$QAbstractItemView$EditKeyPressed
view$setEditTriggers(trigger_flag)
view$verticalHeader()$setHidden(TRUE)
view$horizontalHeader()$setHidden(TRUE)
```

15.13 Implementing custom views

Thus far, we have discussed the application of `QAbstractItemView` for viewing items in a `QAbstractItemModel`. This is the canonical model/view approach in Qt. The role of a `QAbstractItemView` is to display each item in a model, more or less simultaneously. Sometimes it is useful to view an individual item from a model in a simple widget like a label or even an editing widget, such as a line edit or spin box. For example, a GUI for entering records into a database might want to associate each of its widgets with a column in the model, one row at a time.

The `QDataWidgetMapper` class facilitates this by associating a column (or row) in a model with a property on a widget. By default, the *user*

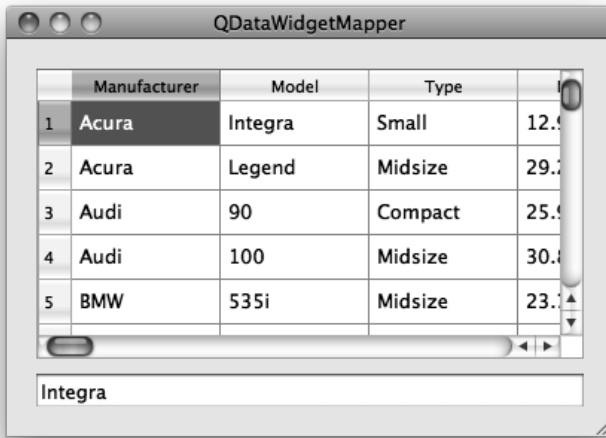


Figure 15.8: The `QDataWidgetMapper` maps the cell value in a column to a property of one or more widgets. Here the line-edit widget is synchronized with the `Model` of the selected row.

property is selected. The user property is marked as the primary user-facing property of a widget; there is only one per class. An example is the `text` property on a `QLineEdit`.

Example 15.4: Mapping selected model items to a text entry

We will demonstrate `QDataWidgetMapper` by displaying a table view of the `Cars93` data set, along with a label. When a row is selected, the `Model` name of the record will be displayed in the label. First, we establish the mapping:

```
data(Cars93, package="MASS")
model <- qdataFrameModel(Cars93, editable=names(Cars93))
mapper <- Qt$QDataWidgetMapper()
mapper$setModel(model)
##
label <- Qt$QLineEdit()
mapper$addMapping(label, 1)
```

The `addMapping` establishes a mapping between the view widget and the 0-based column index in the model. The method prefix is `add` rather than `set`, as more than one mapping is possible.

Next, we construct a table view and establish a handler that changes the current row of the data mapper upon selection:

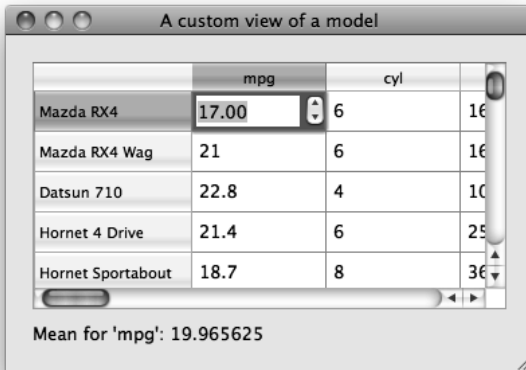


Figure 15.9: Using a label as a custom view. In this case, when the editing is committed, the label is updated to reflect the new mean.

```
table_view <- Qt$QTableView()
table_view$setModel(model)
qconnect(table_view$selectionModel(), "currentRowChanged",
         function(cur,prev) mapper$setCurrentIndex(cur$row()))
```

Finally, we lay out our GUI (Figure 15.8):

```
window <- Qt$QWidget()
layout <- Qt$QVBoxLayout()
window$setLayout(layout)
layout$addWidget(table_view)
layout$addWidget(label)
```

Now, let us consider a different problem: summarizing or aggregating multiple model items, such as an entire column, and displaying the result in a widget. For example, a label might show the mean of a column, and the label would be updated as the model changed. The `QDataWidgetMapper` is not appropriate for this class, as it is limited to a one-to-one mapping between a model item and a widget at any given time. The next example proposes an ad hoc solution to this.

Example 15.5: A label that updates as a model is updated

This example shows how to create an aggregating view for a table model. We will subclass `QLabel` to create a widget (Figure 15.9) that is synchronized to display the mean value of a given column.

In the constructor we define a label property and call our `setModel` method:

```
qsetClass("MeanLabel", Qt$QLabel,
         function(model, column = 0, parent = NULL)
         {
           super(parent)
           this$model <- model
           this$column <- column
           updateMean() # initialize text
           qconnect(model, "dataChanged",
                   function(top_left, bottom_right) {
                     if (top_left$column() <= column &&
                         bottom_right$column() >= column)
                       updateMean()
                   })
         })
```

Whenever the data in the model changes, we want to update the display of the mean value. In the above we call this private method to perform the update:

```
qsetMethod("updateMean", MeanLabel, function() {
  if(is.null(model)) {
    text <- "No model"
  } else {
    DF <- qdataFrame(model)
    colname <- colnames(DF)[column + 1L]
    text <- sprintf("Mean for '%s': %s", colname,
                    mean(DF[,colname]))
  }
  this$text <- text
}, access="private")
```

To demonstrate the use of our custom view, we put it in a simple GUI along with an editable data-frame view. When we edit the data, the text in our label is updated accordingly.

```
model <- qdataFrameModel(mtcars, editable = colnames(mtcars))
table_view <- Qt$QTableView()
table_view$setModel(model)
table_view$setEditTriggers(Qt$QAbstractItemView$DoubleClicked)
##
mean_label <- MeanLabel(model)
##
window <- Qt$QWidget()
layout <- Qt$QVBoxLayout()
window$setLayout(layout)
layout$addWidget(table_view)
layout$addWidget(mean_label)
```

15.14 Viewing and editing text documents

Multiline text is displayed and edited by the `QTextEdit` widget, which is the view and controller for a `QTextDocument` model. The model may be shared among many different views, allowing for synchronized buffers.

`QTextEdit` supports both plain and rich text in HTML format, including images, lists and tables. Applications that display only plain text may be better served by `QPlainTextEdit`, which is faster due to a simpler layout algorithm. `QPlainTextEdit` is otherwise equivalent to `QTextEdit` in terms of API and functionality, so we will focus our discussion on `QTextEdit`, with little loss of generality.

Constructor Here, we create a `QTextEdit` instance and populate it with some text. Although the text is actually stored in a `QTextDocument` instance, it is usually sufficient for us to interact with the `QTextEdit` directly:

```
text_edit <- Qt$QTextEdit()
```

The underlying `QTextDocument` instance can be set by the `setDocument` method but need not be, as one is created on construction.

Adding text to the document can be done easily through the slot `setPlainText`, which replaces the existing text; or the slot `append`, which appends the text as a new paragraph to the end of the buffer.

```
text_edit$setPlainText("The quick brown fox")
text_edit$append("jumped over the lazy dog")
```

As described in its manual page, the widget works on paragraphs and characters, a paragraph being a formatted string, word-wrapped to fit into the width of the widget. For plain text, new lines signify paragraphs.

To return the contents of the model as text, the `toPlainText` method is available:

```
text_edit$toPlainText()
```

```
[1] "The quick brown fox\njumped over the lazy dog"
```

The hard line break `\n` is present, as `append` created a new paragraph.

When text is added to a buffer, it can be undone through the `undo` slot. There are also `redo` to reverse the decision and `undoAvailable` and `redoAvailable` to check for the possibility of each action.

HTML support Instead of plain text, we can also add and insert HTML formatted text for display. The slots `setHTML` and `append` can be used. The `toPlainText` method will return the text with markup stripped off, whereas `toHtml` will return the source HTML of the page.

The text cursor To manage selections, insert special objects like tables and images, or apply the full range of formatting options, it is necessary to interact with a text cursor object, of class `QTextCursor`. Here, we obtain the user-visible cursor and move it to the end of the document:

```
n <- nchar(text_edit$toPlainText())
cursor <- text_edit$textCursor()
cursor$setPosition(n)
text_edit$setTextCursor(cursor)
```

Manipulating the cursor object does not actually modify the location and parameters of the cursor on the screen. We need to set the modified cursor object on the `QTextEdit` explicitly through its `setTextCursor` method. This behavior is often convenient, because it allows us to modify arbitrary parts of the document, without affecting the user cursor. For example, we could insert a 32- x -32-pixel image at the beginning:

```
cursor$setPosition(0) # move to beginning
style <- Qt$QApplication$style()
icon <- style$standardIcon(Qt$QStyle$SP_DialogOkButton)
sz <- QSize(32L,32L)
anImage <- icon$pixmap(icon$actualSize(sz))$toImage()
cursor$insertImage(anImage)
```

In the above, we moved the cursor through its `setPosition` method. If the document is viewed as a single string of characters, the position i would refer to the space between the i th and $i + 1$ st character, 0 being the initial point in the document.

The motion of the cursor is described by the enumeration `QTextCursor$MoveOperation`, with several values such as "Start", "End", "StartOfLine", "EndOfLine", "StartOfWord", "EndOfWord" etc.

For example, to move the cursor to the start of the second line, we could do:

```
cursor <- text_edit$textCursor()
cursor$movePosition(Qt$QTextCursor$Start) # MoveAnchor default
cursor$movePosition(Qt$QTextCursor$Down) # down one line
text_edit$setTextCursor(cursor)
```

Selection Selection is a component of the `QTextCursor` state. For plain text, the selected text is returned by the `selectedText` method:

```
text_edit$textCursor()$selectedText() # no current selection
```

NULL

The NULL value indicates that the user has not selected any text.

A text cursor has an anchor position in addition to its position. The selection is the text between the two. When moving the cursor through

its `movePosition` method, we can choose to move or keep the anchor in place. Normally, the anchor and cursor are at the same position. To make a selection programatically, we move the cursor independently of its anchor. The `QTextCursor$MoveMode` enumeration with values "MoveAnchor" and "KeepAnchor" can be specified to `movePosition` to control this. Here we set the selection to include the first three words of the text in the second line. We have:

```
cursor <- Qt$QTextCursor(text_edit$document())
cursor$movePosition(Qt$QTextCursor$Start)      # as before
cursor$movePosition(Qt$QTextCursor$Down)      # moves anchor
cursor$movePosition(Qt$QTextCursor$WordRight, # anchor fixed
                    Qt$QTextCursor$KeepAnchor, 3)
text_edit$setTextCursor(cursor)
```

The 3 specified to `movePosition` calls the action three times.

Now our selection yields:

```
cursor$selectedText()
```

```
[1] "jumped over the "
```

Signals There are several different signals emitted by `QTextEdit` instances: `textChanged`, when the text changes; `cursorPositionChanged`, when the cursor position changes; and `selectionChanged`, when the selection changes (according to the user-visible cursor). For the latter, the `copyAvailable` signal is largely equivalent, except it passes a Boolean argument indicating whether the selection is non-empty.

```
qconnect(text_edit, "textChanged", function() {
  message("Text has changed to", text_edit$toPlainText())
})
##
qconnect(text_edit, "cursorPositionChanged", function() {
  message("Cursor has changed. It is now in position",
         text_edit$textCursor()$position())
})
##
qconnect(text_edit, "selectionChanged", function() {
  message("text: ", text_edit$textCursor()$selectedText())
})
```

Formatting properties By default, the widget will wrap text as entered. For use as a code editor, this is not desirable. The `lineWrapMode` property takes values from the enumeration `QTextEdit::LineWrapMode` to control this:

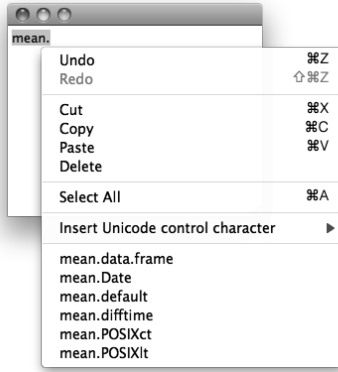


Figure 15.10: Context menu showing completion candidates for the token "mean" taken from the current selection.

```
text_edit$lineWrapMode <- Qt$QTextEdit$NoWrap
```

We can fix the wrapping at a certain number of characters by using a wrap mode of `FixedColumnWidth` and setting the count through `lineWrapColumnOrWidth`.

The `setAlignment` method aligns the current paragraph (the one with the cursor) with values from `Qt::Alignment`.

Character attributes The widget keeps track of a current set of formatting options in an object of class `QTextCharFormat`. The text-edit methods `setCurrentFont`, `setFontFamily`, and `setFontWeight`, among others, modify the current settings. If called when there is a selection, the change will be applied to the selection in addition to any new text.

Syntax highlighting The text-edit widget supports syntax highlighting through the `QSyntaxHighlighter` class. To implement a specific highlighting rule, we must subclass `QSyntaxHighlighter` and override the `highlightBlock` method to apply highlighting. This is of somewhat special interest, so we will not give an example. For a syntax-highlighting R-code viewer and editor, see `qeditor` in the `qtutils` package.

Searching The `find` method will search for a given string and adjust the cursor to select the match. For example, we can search through a standard typesetting string starting at the cursor point for the common word "qui" as follows:

```
text_edit <- Qt$QTextEdit(LoremIpsum) # some text
text_edit$find("qui", Qt$QTextDocument$FindWholeWords)
```

```
[1] TRUE
```

```
text_edit$textCursor()$selection()$toPlainText()
```

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

The second parameter to `find` takes a combination of flags from `QTextDocument::FindFlag`, with values "FindBackward", "FindCaseSensitively", and "FindWholeWords".

Context menus As we introduce in Section 16.3 of Chapter 16, we can enable a dynamic context menu on a widget by overriding the `contextMenuEvent` virtual. For our demonstration, we aim to list candidate completions based on the currently selected text:

```
qsetClass("QTextEditWithCompletions", Qt$QTextEdit)
##
qsetMethod("contextMenuEvent", QTextEditWithCompletions,
           function(event)
             {
               menu <- this$createStandardContextMenu()
               if(this$textCursor()$hasSelection()) {
                 selection <- this$textCursor()$selectedText()
                 completions <- utils:::matchAvailableTopics(selection)
                 completions <- setdiff(completions, selection)
                 if(length(completions) > 0 && length(completions) < 25) {
                   menu$addSeparator() # add actions
                   sapply(completions, function(completion) {
                     action <- Qt$QAction(completion, this)
                     qconnect(action, "triggered", function(checked) {
                       insertPlainText(completion)
                     })
                     menu$addAction(action)
                   })
                 }
               }
               menu$exec(event$globalPos())
             })
text_edit <- QTextEditWithCompletions()
```

The `createStandardContextMenu` method returns the base context menu, including functions like copy and paste. We add an action for every possible completion (Figure 15.10). Triggering an action will paste the completion into the document replacing the current selection with the chosen completion candidate.

This page intentionally left blank

Qt: Application Windows

Many applications have a central window that typically contains a menu bar, toolbar, an application-specific area, and a status bar at the bottom. This is known as an application window and is implemented by the `QMainWindow` widget. Although any widget in Qt might serve as a top-level window, `QMainWindow` has explicit support for a menu bar, toolbar and status bar, and also provides a framework for dockable windows.

To demonstrate the `QMainWindow` framework, we will create a simple spreadsheet application (Figure 16.1). First, we construct a `QMainWindow` object:

```
main_window <- Qt$QMainWindow()
```

The region between the toolbar and status bar, known as the central widget, is completely defined by the application. We wish to display a spreadsheet, i.e., an editable table:

```
data(mtcars)
model <- qdataFrameModel(mtcars, editable = TRUE)
table_view <- Qt$QTableView()
```

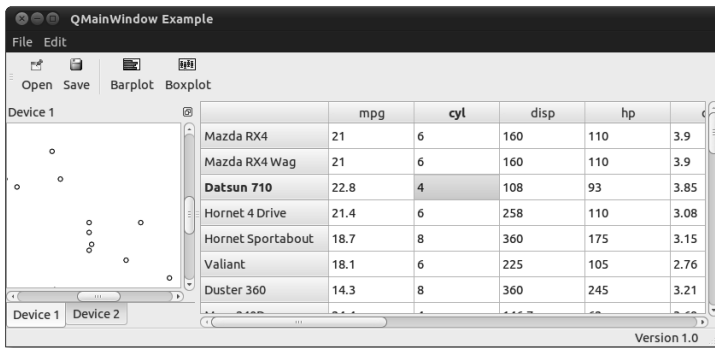


Figure 16.1: An example of a GUI with menu, tool, and status bars, along with dockable windows, constructed using a `QMainWindow` instance.


```
table_view$setModel(model)
main_window$setCentralWidget(table_view)
```

We will continue by adding a menu bar and toolbar to our window. This depends on an understanding of how Qt represents actions.

16.1 Actions

The buttons in the menu bar and toolbar, as well as other widgets in the GUI, might share the same action. Thus, it is sensible to separate the definition of an action from any individual control. An action is defined by 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. The properties of a `QAction` include the label text, icon, `toolTip`, `statusTip`, keyboard shortcut, and whether the action is enabled.

We construct an action for opening a file:

```
open_action <- Qt$QAction("Open", main_window)
```

The label text is passed to the constructor along with the parent window. We can specify additional properties, such as the text to display in the status bar when the user moves the mouse over a widget proxying the action:

```
open_action$statusTip <- "Load a spreadsheet from a CSV file"
```

We could also set an icon:

```
style <- Qt$QApplication$style()
open_action$icon <-
  style$standardIcon(Qt$QStyle$SP_DialogOpenButton)
```

Actions emit a triggered signal when activated. The application should connect to this signal to implement the command behind the action:

```
qconnect(open_action, "triggered", function() {
  filename <- Qt$QFileDialog$getOpenFileName()
  table_view$model <-
    qdataFrameModel(read.csv(filename), editable = TRUE)
})
```

Toggle and radio actions An action may have a Boolean state, i.e., it may be checkable. This is controlled by the `checkable` property. When a checkable action is triggered, its state is toggled and the current state is passed to the trigger handler. For example, we could have an action to toggle whether the spreadsheet will be saved on exit:

```
save_on_exit_action <- Qt$QAction("Save on exit", main_window)
save_on_exit_action$checkable <- TRUE
```

The checked property reports whether the action has been checked or not. For this type of action, we would query this on exit. For other implementations, where the action should be enacted immediately, we would connect to the changed signal.

A checkable action in isolation behaves much like a check button. If checkable actions are placed together into a QActionGroup, the default behavior is such that only one is checked at once, analogous to a set of radio buttons. We could have an action for controlling the justification mode for the text entry:

```
just_group <- Qt$QActionGroup(main_window)
just_action <- list()
just_action$left <- Qt$QAction("Left Align", just_group)
just_action$right <- Qt$QAction("Right Align", just_group)
just_action$center <- Qt$QAction("Center", just_group)
sapply(just_action, function(action) action$checkable <- TRUE)
```

Here, we connect to each action's changed signal to broadcast which button was pressed.

```
sapply(just_action, function(action)
  qconnect(action, "changed", function() {
    button_number <-
      which(sapply(just_action, '[', "checked"))
    message("Button ", button_number, " was depressed")
  })
)
```

We could also connect to the triggered signal of the action group. The callback is passed the action object.

```
qconnect(just_group, "triggered", function(action) {
  message(action$text)
})
```

Keyboard shortcuts Every platform has a particular convention for mapping key presses to typical actions. Qt abstracts some common commands via the QKeySequence::StandardKey enumeration, a member of which may refer to multiple key combinations, depending on the command and the platform. We assign the appropriate shortcuts for our “Open” action:

```
open_action$setShortcut(Qt$QKeySequence(Qt$QKeySequence$Open))
```

Whenever the window has focus and the user presses the conventional key sequence, such as Ctrl-O on Windows, our action will be triggered. It is important not to confuse this shortcut mechanism with mnemonics, which

are often indicated by underlining a letter in the label text of a menu item. A mnemonic is active only when the parent menu is active. Mnemonics are disabled by default on Windows and Mac installations of Qt and thus are not covered here.

16.2 Menu bars

Applications often support too many actions to display them all at once. The typical solution is to group the actions into a hierarchical system of menus. The menu bar is the top-level entry point to the hierarchy. The placement of the menu bar depends on the platform. On Mac OS X, applications share a menu bar area at the top of the screen. On other platforms, the menu bar is typically found at the top of the main window for the application.

We create an instance of `QMenuBar` and set it for the main window:

```
menubar <- Qt$QMenuBar()
main_window$setMenuBar(menubar)
```

A `QMenuBar` instance is a container for `QMenu` objects, which represent the submenus. We create a `QMenu` for the “File” and “Edit” menus and add them to the menu bar:

```
file_menu <- Qt$QMenu("File")
menubar$addMenu(file_menu)
edit_menu <- Qt$QMenu("Edit")
menubar$addMenu(edit_menu)
```

To each `QMenu` we may add:

1. an action through the `addAction` method,
2. a separator through `addSeparator`, or
3. nested submenus through the `addMenu` method.

We demonstrate each of these operations by populating the “File” and “Edit” menus:

```
file_menu$addAction(open_action)
file_menu$addSeparator()
file_menu$addAction(save_on_exit_action)
file_menu$addSeparator()
quit_action <- file_menu$addAction("Quit")
just_menu <- edit_menu$addMenu("Justification")
sapply(just_action, just_menu$addAction)
```

In the above, we take advantage of the convenient overloads of `addAction` and `addMenu` that accept a string title and return a new `QAction` or `QMenu`, respectively.

16.3 Context menus

Sometimes, an action pertains to a single widget or portion of a widget, instead of the entire application. In such cases, the menu bar is an inappropriate container. An alternative is to place the actions in a menu specific to their context. This is known as a context menu. The precise user action that displays a context menu depends on the platform. It commonly suffices to click the right mouse button while the pointer is over the widget. The simplest approach to providing a context menu involves two steps. First, add the desired actions to the widget:

```
sort_menu <- Qt$QMenu("Sort by")
sapply(colnames(qdataFrame(model)), sort_menu$addAction)
table_view$addAction(sort_menu$menuAction())
```

Second, we configure the widget to display a menu of the actions when a context menu is requested:

```
table_view$contextMenuPolicy <- Qt$Qt$ActionsContextMenu
```

The simple approach is appropriate in most cases. One limitation, however, is that the actions need to be defined prior to the context menu request. For example, if we allowed adding and removing columns in the spreadsheet, we would need to adjust the actions in the sort context menu. Another example is a code-entry widget, where a pop-up window could list possible code completions. In Section 15.14, we implement a text-completion pop-up in an override of the `contextMenuEvent` virtual method.

If subclassing is undesirable, we could change the context menu policy to `CustomContextMenu` and connect to the signal `customContextMenuRequested`:

```
showCompletionPopup <- function(event, edit) {
  popup <- Qt$QMenu()
  completions <- utils::matchAvailableTopics(ed$text)
  completions <- head(completions, 10) # trim if large
  sapply(completions, function(completion) {
    action <- popup$addAction(completion)
    qconnect(action, "triggered",
              function(...) edit$setText(completion))
  })
  popup$popup(edit$mapToGlobal(qpoint(0L,0L)))
}
```

```
##
edit <- Qt$QLineEdit()
edit$contextMenuPolicy <- Qt$Qt$CustomContextMenu
qconnect(edit, "customContextMenuRequested",
          showCompletionPopup, user.data = edit)
```

16.4 Toolbars

The toolbar manages a compact layout of frequently executed actions, so that the actions are readily available to the user without consuming an excessive amount of screen space. We create a `QToolBar` and add it to our main window:

```
toolbar <- Qt$QToolBar()
main_window$addToolBar(toolbar)
```

The main window places the toolbar into a toolbar area, which might contain multiple toolbars. It is possible, by default, for the user to rearrange the toolbars by clicking and dragging with the mouse. If the toolbar is pulled out of the toolbar area, it will become an independent window.

To add items to a toolbar we might call

- `addAction` to add an action,
- `addWidget` to embed an arbitrary widget, or
- `addSeparator` to place a separator between items.

We create each action, set its icon (the `getIcon` function is not shown), and store it in a list for ease of manipulation at a later time in the program:

```
file_actions <- list()
file_actions$open <- Qt$QAction("Open", main_window)
file_actions$open$setIcon(getIcon("open"))
file_actions$save <- Qt$QAction("Save", main_window)
file_actions$save$setIcon(getIcon("save"))
plot_actions <- list()
plot_actions$barplot <- Qt$QAction("Barplot", main_window)
plot_actions$barplot$setIcon(getIcon("barplot"))
plot_actions$boxplot <- Qt$QAction("Boxplot", main_window)
plot_actions$boxplot$setIcon(getIcon("boxplot"))
```

Finally, we add the actions to the toolbar, with a separator between the file actions and plot actions:

```
sapply(file_actions, toolbar$addAction)
toolbar$addSeparator()
sapply(plot_actions, toolbar$addAction)
```

QToolBar will display actions as buttons, and the precise configuration of the buttons depends on the toolbar style. For example, the buttons might display only text, only icons, or both. By default, only icons are shown. We instruct our toolbar to display an icon, with the label underneath:

```
toolbar$setToolButtonStyle(Qt$Qt$ToolButtonTextUnderIcon)
```

By default, toolbars pack their items horizontally. Vertical packing is also possible; see the `orientation` property.

16.5 Status bars

Main windows reserve an area for a status bar at the bottom of the window. The status bar is used to display messages about the current state of the program, as well as any status tips assigned to actions.

A status bar is an instance of the `QStatusBar` class. We create one and add it to our window:

```
statusbar <- Qt$QStatusBar()
main_window$setStatusbar(statusbar)
```

There are three types of messages in a status bar:

- 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 and appears at the far right.

In addition to messages, we can embed widgets into the status bar.

For example, we could communicate a temporary message when a data set is loaded:

```
statusbar$showMessage("Load complete", 1000)
```

The second argument above is optional and indicates the duration of the message in milliseconds. If not specified, the message must be explicitly cleared with `clearMessage`.

Normal and permanent messages must be placed into a `QLabel`, which is then added to the status bar like any other widget:

```
statusbar$addWidget(Qt$QLabel("Ready"))
statusbar$addPermanentWidget(Qt$QLabel("Version 1.0"))
```

16.6 Dockable widgets

QMainWindow supports window docking. There is a *dock area* for each of the four sides of the window (top, bottom, left, and right). If a widget is assigned to a dock area, the user may, by default, drag the widget between the docking areas. If multiple widgets are placed into the same area, they are grouped into a tabbed notebook. Dragging a docked widget to a location outside of a dock area will convert the widget into a top-level window.

For example, we could add an R-graphics device as a dockable widget. The first step is to wrap the widget in a QDockWidget:

```
library(qtutils)
device <- QT()
dock <- Qt$QDockWidget("Device 1")
dock$setWidget(device)
```

The string passed to the QDockWidget constructor is an optional label/title for the docked window.

By default, the dock widget is closable, movable and floatable. This is adjustable through the features property. For example, we could disable closing of the graphics device:

```
dock$features <- Qt$QDockWidget$DockWidgetMovable |
               Qt$QDockWidget$DockWidgetFloatable
```

The allowedAreas property specifies the valid docking areas for a dock widget. By default, all are allowed.

After configuring the dock widget, we add it to the main window, in the left docking area:

```
main_window$addDockWidget(Qt$Qt$LeftDockWidgetArea, dock)
```

A second graphics device could be added with the first, on a separate page of a tabbed notebook:

```
device2 <- QT()
dock2 <- Qt$QDockWidget("Device 2", device2)
main_window$tabifyDockWidget(dock, dock2)
```

To make dock2 a top-level window instead, we could set the floating property to "TRUE":

```
dock2$floating <- TRUE
```

Part IV

The tcltk Package

This page intentionally left blank

Tcl/Tk: Overview

“Tool Command Language” (Tcl) is a scripting language and interpreter of that language. Originally developed in the late 1980s by John Ousterhout as a “glue” to combine two or more complicated applications, it evolved over time to find use not just as middleware, but also as a stand-alone development tool.

Tk is an extension of Tcl that provides GUI components through Tcl. Tk was first developed in 1990, again by John Ousterhout. It quickly found widespread usage, as at the time 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 many users.

There are a large number of language bindings available for Tk, including Perl, Python, Ruby, and, through the `tcltk` package, R. The `tcltk` package was developed by Peter Dalgaard and has been included with base R since version 1.1.0. Since then, the package has been used in a number of GUI projects for R, most notably the Rcmdr GUI. The `tcltk2` package provides additional bindings and bundles in some useful external TCL code. Our focus here is limited to the base `tcltk` package.

Tk had a major change between versions 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, corresponds to the newer themed variant of the widget. We assume the Tk version is 8.5 or higher, as this was a major step forward.¹

Despite Tk’s limitations as a graphical toolkit as compared to GTK+ or Qt, the Tk libraries are widely used for R GUIs. R for Windows has been bundled with the necessary Tk version for years, so there are no installation issues for that platform. For Linux users, it is typically trivial to install the newer libraries, and for Mac OS X users, the provided binary installations include the newer Tk libraries.

¹In fact, we assume version 8.5.8, which was the release accompanying R for Windows version 2.13.1.

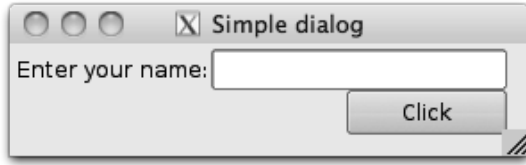


Figure 17.1: A simple dialog to collect a name for later use illustrates three basic widgets: a label, entry widget, and button.

Tk has a well-documented API^[10] (www.tcl.tk/man/tcl8.5/). There are also several books to supplement. We consulted the one by Welch, Jones and Hobbs^[1] often in the development of this material. The online sample chapter on geometry management of Walsh^[13] was perused, as it provides a thorough discussion of that topic. In addition, the Tk Tutorial of Mark Roseman^[9] (www.tkdcs.com/tutorial) provides much detail. R-specific documentation includes two excellent *R News* articles and a proceedings paper^{[3][5][4]} by Peter Dalgaard, the package author. A set of examples by James Wettenhall^[14] 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^[6] by John Fox, the package author.

17.1 A first example

In this chapter we give an overview of Tk and R's interface to it through the `tcltk` package using the following small example of a dialog to collect a name and echo back a message (Figure 17.1). In subsequent chapters we give more detail on the various widgets provided by Tk.

```
library(tcltk)
##
```

[10] Tcl Core Team. <http://www.tcl.tk/man/tcl8.5/>.

[1] Jeffrey Hobbs Brent B. Welch, Ken Jones. *Practical Programming in Tcl and Tk*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 2003.

[13] Nancy Walsh. *Learning Perl/Tk: Graphical User Interfaces with Perl*. O'Reilly, first edition, January 1999. <http://oreilly.com/catalog/9781565923140>.

[9] Mark Roseman. <http://www.tkdcs.com/tutorial/>.

[3] Peter Dalgaard. A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31, September 2001.

[5] Peter Dalgaard. Changes to the R-Tcl/Tk package. *R News*, 2(3):25–27, December 2002.

[4] Peter Dalgaard. The R-Tcl/Tk interface. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, 2001. ISSN 1609-395X.

[14] James Wettenhall. <http://bioinf.wehi.edu.au/~wettenhall/RTclTkExamples/>.

[6] John Fox. Extending the R Commander by “plug-in” Packages. *R News*, 7(3):46–52, December 2007.

```

window <- tktoplevel()
tkwm.title(window, "Simple dialog")
##
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both")
##
nested_frame <- ttkframe(frame); tkpack(nested_frame)
##
label <- ttklabel(nested_frame, text = "Enter your name:")
tkpack(label, side = "left")
##
text_var <- tclVar("")
entry <- ttkentry(nested_frame, textvariable = text_var)
tkpack(entry)
##
button_frame <- ttkframe(frame)
tkpack(button_frame, anchor = "ne")
button <- ttkbutton(button_frame, text = "Click")
tkpack(button, side = "right")
##
handler <- function() {
  msg <- sprintf("Hello %s", tclvalue(text_var))
  print(msg)
}
tkconfigure(button, command = handler)

```

In the above, the first block defines a top-level window and the second an underlying frame container. We then define and place three widgets – a label, entry widget, and button – into a frame. Finally, we add a callback to respond when the button is clicked.

17.2 Interacting with Tcl

As the example above makes clear, using `tcltk` does not necessarily require knowing anything about the underlying Tk or Tcl workings, though it can be useful to have a rough sense of these technologies and how `tcltk` interfaces with them. As such, we give a quick overview.

Although both are scripting languages, the basic syntax of Tcl is unlike that of 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, in which 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 it 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, we 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") # prints to stdout

some text

.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`). The `.Tcl` function can be used to read in Tcl scripts as with `.Tcl("source filename")`. This allows arbitrary Tcl scripts to run within an R session. Tcl packages can be read in with `tclRequire`.²

²The add-on package `tcltk2` uses both techniques to enhance the base `tcltk` package with some open-source Tk extensions.

The tclObj class The tcltk package creates objects with a few different classes, tclObj being the primary one (tclVar and tkwin are two other important ones). The tclObj objects print with the leading <Tcl>. The string representation of objects of class tclObj is returned by tclvalue or by coercion through the as.character function. These two differ in how they treat spaces and new lines. Conversion to vectors of mode character, double, integer, and logical is possible, though, in general, direct conversion of complicated Tcl expressions is not supported. We can create objects of this class through as.tclObj.

Convenience functions 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 configure the text property for a label object (.label) would look like

```
% .label configure -text "new text"
```

The tcltk package provides a corresponding function tkconfigure. The above would be done in an R-like way as (assuming label is a label object):

```
tkconfigure(label, text = "new text")
```

The Tk API for ttklabel's configure subcommand is

pathName **configure** *?option? ?value option value ...?*

The *pathName* is the ID of the label widget. This can be found from the object label above, in label\$ID, or in some cases is a return value of some other command call. In the Tk documentation, paired question marks indicate optional values. In this case, we 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; or possibly more than one at a time. For commands such as configure, there will usually correspond a function in R of the same name with a tk prefix, as in the case tkconfigure.

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 the reader to the original Tk documentation; understanding this is important for reading the API.) However, if such a function shortcut is present, we will typically use the shortcut when we illustrate code.

Some subcommands have further subcommands. An example is with setting the selection. In the R function, the second command is appended with a dot, as in tkselection.set. (There are a few necessary 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, call the interpreter, and then return an object of class `tclObj`. Behind the scenes it

- turns an R object, `widget`, into the `pathName` above (using its ID component)
- passes along strings as subcommands (`configure`)
- converts R `key=value` pairs into `-key value` options for Tcl; named arguments are for only the `-key value` expansion, we follow the Tcl language and call the arguments “options” in the following
- adjusts any callback functions allowing, R functions and expressions to be called

The `tcl` function uses position to create its command. The order of the subcommands needs to match that of the Tk API, so although it is true that often the R object is first, this is not always the case.

17.3 Constructors

In this chapter, we will stick to a few basic widgets – labels, entry widgets, and buttons – to illustrate the usage of `tcltk`, leaving for later more detail on containers and widgets.

Unlike GTK+, say, the construction of widgets in `tcltk` is linked to the widget hierarchy. Tk widgets are constructed as children of a parent object, 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. (This is `.TkRoot` in R.) 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 was done in the example with:

```
window <- tktoplevel()
```

Top-level windows will be explained in more detail in Chapter 18.

Other widget constructors require that a parent widget be specified as the first argument of the constructor. A typical invocation was given in the example.

```
label <- ttklabel(nested_frame, text = "Enter your name:")
```

Options The first argument of a widget constructor is the parent container; subsequent arguments, given as key=value pairs, are used to specify the options for the constructor. The Tk API lists these options along with their descriptions.

For a simple label, the following options are possible: `anchor`, `background`, `font`, `foreground`, `justify`, `padding`, `relief`, `text`, and `wraperlength`. This is in addition to the standard options `class`, `compound`, `cursor`, `image`, `state`, `style`, `takefocus`, `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 meanings are clear from their name. The main option, `text`, takes a character string. The label will be multiline if it contains new line characters. 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 (`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"`, and `"sunken"`.

The functions `tkconfigure`, `tkcget` Option values can be set through the constructor or adjusted afterwards by `tkconfigure`. A listing (in Tcl code) of possible options that can be adjusted can be seen by calling `tkconfigure` with just the widget as an argument.

```
head(as.character(tkconfigure(label)))           # 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 ways: using the Tk style of a leading dash or using the R convention that `NULL` values mean to return the value and not set it.

```
tkcget(label, "-text")                          # retrieve text property

<Tcl> Enter your name:

tkcget(label, text = NULL)                       # alternate syntax

<Tcl> Enter your name:
```


Coercion to character As mentioned, the `tclObj` objects can be coerced to characters in two ways. The conversion through `as.character` breaks the return value along white space:

```
as.character(tkcget(label, text = NULL))
```

```
[1] "Enter" "your" "name:"
```

whereas conversion by the `tclvalue` function does not:

```
tclvalue(tkcget(label, text = NULL))
```

```
[1] "Enter your name:"
```

The `tkwidget` function

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). For example,

```
str(button)
```

```
List of 2
 $ ID : chr ".1.1.2.1"
 $ env:<environment: 0x1032edd40>
 - 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 hierarchy of the object. This value is generated behind the scenes by the `tcltk` package using numeric values to keep track of the hierarchy. The `env` component contains an environment that keeps a count of the subwindows, the parent window, and any callback functions. This helps ensure that any copies of the widget refer to the same object.^[4] As the construction of a new widget requires the ID and environment of its parent, the first argument to `tkwidget` (and hence any constructor), `parent`, must be a `tkwin` object, not simply its character ID, as is possible for the `tcl` function.

Geometry managers

In the example we saw several calls to `tkpack`. For example,

```
tkpack(frame, expand = TRUE, fill = "both")
tkpack(label, side = "left")
tkpack(entry)
tkpack(button_frame, anchor = "ne")
```

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, `tkpack` and `tkgrid`, in Chapter 18.

The `tkpack` command packs the widgets into the parent container in a boxlike manner. The example shows various arguments that adjust the position of the child component and how space is to be allocated when an excess of space is present.

Tcl variables

For the button and label widgets in our example, their text property is configured through calls to their constructors. Many widgets allow an alternative way to specify one or two important properties using an independent Tcl variable.

In the call to `ttkentry` in the example we had:

```
text_var <- tclVar("")
entry <- ttkentry(g, textvariable = text_var)
```

The first line defines a new object of class `tclVar` which is used for the `textvariable` option when defining the entry widget. This variable is 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 Tcl variable can be used with more than one widget, allowing a simple form of synchronization.

The basic functions involved are `tclVar` to create a Tcl variable, `tclvalue` to get the assigned value, and `tclvalue<-` to modify the value.

```
tclvalue(text_var) <- "Somebody's name"
tclvalue(text_var)
```

```
[1] "Somebody's name"
```

Tcl variables have a unique identifier, returned by `as.character`:

```
as.character(text_var)
```

```
[1] ":::RTcl1"
```

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, the usage of Tcl variables is recommended. One disadvantage is that in a callback, the variable is not passed to the callback and can't be recovered from the object itself. Hence, it must be found through R's scoping rules. (In Section 19.2 we show a work-around.)

The package also provides the function `tclArray` to store an array of Tcl variables. The usual list methods, `[[` and `$`, and their forms for assignment are available for arrays, but values are referred to only by name, not index:

```
x <- tclArray()           # no init
x$one <- 1; x[[2]] <- 2   # $<- and [[<-
x[[1]]                   # no match by index

NULL

names(x)                  # the stored names

[1] "2" "one"

x[['2']]                  # match by name, not index

<Tcl> 2
```

Commands

In the definition of the button we saw:

```
button <- ttkbutton(button_frame, text = "Click")
#
handler <- function() {
  msg <- sprintf("Hello %s", tclvalue(text_var))
  print(msg)
}
tkconfigure(button, command = handler)
```

Button widgets are used to initiate some action or command, and the `command` option is used to specify this. This may be given as a function or expression, though we illustrate only the former. The command is invoked by clicking and releasing the mouse on the button, by pressing the space bar when the button has the focus, or by calling the widget's `ttkbutton` invoke subcommand.

The `command` option is available for many widgets, but it is not the only means to invoke a function call, as Tk also allows us to bind to various types of events (e.g., button clicks). More on callbacks in `tcltk` will be explained in Section 17.4.

Themes

As mentioned, the newer themed widgets have a style that determines how they are drawn based on the state of the widget. 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



Figure 17.2: Comparison of themed versus non-themed dialogs. The non-themed one (right) one does not use an inner `ttkframe`, and in addition to not having padding, has mismatched colors.

and easier to maintain the code. A collection of styles makes up a theme. The available themes depend on the system. The default theme allows the GUI to have the native look and feel of the operating system. (This was definitely not the case for the older Tk widgets.)

In our example, the top-level window has a frame immediately packed inside of it through the commands:

```
window <- tkoplevel()
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both")
```

The arguments to `tkpack` are given so that the frame, `frame`, will expand and fill all the space allocated by the top-level window. As the top-level window is not a themed widget, not doing this can leave odd-looking effects, see Figure 17.2.

There is no built-in command to return the theme, so we use `.Tcl` to call the appropriate names subcommand:

```
.Tcl("ttk::style theme names")
```

```
<Tcl> clam alt default classic
```

The `use` sub command is used to set the theme:

```
.Tcl("ttk::style theme use clam")
```

State of themed widgets The themed widgets (those with a `ttk` constructor) have a state to determine which style is to be applied when painting the widget. These states can be adjusted through the `state` command and queried with the `instate` command. For example, to see if button widget `b` has the focus, we have:

```
as.logical(tcl(button, "instate", "focus"))
```

```
[1] FALSE
```

To set a widget to not be sensitive to user input we have:

```
tcl(button, "state", "disabled") # not sensitive
```

```
<Tcl> !disabled
```

The states are bits and can be negated by prefacing the value with !:

```
tcl(button, "state", "!disabled") # sensitive again
```

```
<Tcl> disabled
```

The full list of states is in the manual page for `ttk::widget`.

The writing of themes will not be covered, but in Example 18.5 we show how to create a new style for a button. This topic is covered in some detail in the Tk tutorial by Roseman.

Window properties and state: tkwininfo

For a widget, the function `tkcget` is used to get the values of its options. If it is a themed widget, the `instate` command returns its state values.

To query the values of the containing window of the widget the `tkwininfo` function is used. When widgets are mapped, the “window” they are rendered to has properties, such as class or size. There are a few subcommands provided by `tcltk`, but by no means is this exclusive. Rather, one can pass in the subcommand as an argument to `tkwininfo`. If the subcommand’s 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 class³ of a label is returned by the `class` subcommand:

```
tkwininfo("class", label)
```

```
<Tcl> TLabel
```

The window, in this example `label`, 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", window))
```

```
<Tcl> .4.1 .4.2
```

```
apply(as.character(children), function(i) tkwininfo("class", i))
```

³The class of a widget is more like an attribute and should not be confused with class in the object-oriented sense. The class is used internally for bindings and styles.

```
$ '.4.1'
<Tcl> TButton

$.4.2'
<Tcl> TButton
```

There are several possible subcommands. Here we list a few. The *tkwinfo* geometry sub command returns the location and size of the widget's window in the form width x height + x + y; the subcommands *tkwinfo* height, *tkwinfo* width, *tkwinfo* x, or *tkwinfo* y can be used to return just those parts. The *tkwinfo* exists command returns 1 (TRUE) if the window exists and 0 otherwise; the *tkwinfo* ismapped sub command returns 1 or 0 if the window is currently mapped (drawn); the *tkwinfo* viewable subcommand is similar, except that it checks that all parent windows are also mapped.

For traversing the widget hierarchy, we have available the *tkwinfo* parent subcommand which returns the immediate parent of the component, *tkwinfo* toplevel which returns the ID of the top-level window; and *tkwinfo* children, which returns the IDs of all the immediate child components, if the object is a container, such as a top-level window.

Colors and fonts

Colors and fonts are typically specified through a theme, but at times it is desirable to customize them.

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(label, foreground = "red")
tkconfigure(label, foreground = "#00aa00")
```

To find the hex RGB value, we 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.

In Example 19.2 we show how to modify a style, as opposed to the foreground option, to change the text color in an entry widget.

Fonts Fonts are a bit more involved than colors. Tk version 8.5 made it more difficult to change font properties of individual widgets, by 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 specifying the font properties, we configure the font options using a predefined font name, such as

```
tkconfigure(label, font = "TkFixedFont")
```

Table 17.1: 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

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 17.3. Each theme sets these defaults accordingly.

Using `tkfont.create` The `tkfont.create` function can be used to create a new font, as with the following commands:

```
tkfont.create("our_font", family = "Helvetica", size = 12,
             weight = "bold")
```

```
<Tcl> our_font
```

```
tkconfigure(label, font = "our_font")
```

As font families are system dependent, only "Courier", "Times" and "Helvetica" are guaranteed to be there. A list of an installation's available font families is returned by the function `tkfont.families`. Figure 17.3 shows a display of some available font families on a Mac OS X machine. See Example 20.7 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. (For example, the text widget specifies its height in lines, not pixels.) These sizes can be found using the `tkfont.measure` and `tkfont.metrics`. Although the average text size varies for proportional fonts, the size of the M character is often used.

```
font_measure <- tcl("font", "measure", "TkTextFont", "M")
font_width <- as.integer(tclvalue(font_measure))
```



Figure 17.3: A scrollable frame widget (cf. Example 20.7) showing the available fonts on a system.

```
tmp <- tkfont.metrics("TkTextFont", "linespace" = NULL)
font_height <- as.numeric(tclvalue(tmp))
#
c(width = font_width, height = font_height)
```

```
width height
10      14
```

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. This shared use of resources reduces memory consumption and is an example of the flyweight design pattern.

Images are created by the `tkimage.create` function. 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).⁴ 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 command `tcl("image", "names")` will return all defined image names.) The third argument `file` specifies the graphic file. The basic Tk image command can only show GIF and PPM/PNM images. Unfortunately, not many R devices output in these formats. (The GDD device driver can.) We may need system utilities to convert to the allowable formats or install add-on Tcl packages that can display other formats.

To use the image, one specifies the image name to the `image` option:

```
label <- ttklabel(window, 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 manual page for `photo`, not `image`. For instance, to change the palette so that instead of `fullcolor` only sixteen shades of gray are used to display the icon, We can issue the command

```
tkconfigure "::img::tclLogo", palette = 16)
```

Other commands allow us to scale an image (`copy zoom` and `copy subsample`).

17.4 Events and callbacks

The button widget has the `command` option for assigning a callback, which is invoked when the user clicks the mouse on the button (among other ways). In addition to such commands, we can use `tkbind` to invoke callbacks in response to many other events that the user may initiate. The basic call is `tkbind(tag, event, script)`.

⁴The `tkrplot` package allows a third option, `Rplot`. This package has the high-level command `tkrplot`, but the low-level use of a) calling `.my.tkdev(hscale = 1, vscale = 1)`, b) creating a graphic, and c) creating an image object through `tkimage.create("Rplot", img_name)` will produce a new image object we can use.

The tag

The tag object is more general than just a widget (or its id). 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.

This flexibility makes it easy to create keyboard accelerators. For example, the following mimics the Linux shortcut Control-q to close a window.

```

window <- tkoplevel()
button <- ttkbutton(window, text = "Some widget with focus")
tkpack(button)
tkbind(window, "<Control-q>", function() tkdestroy(window))

```

By binding to the top-level window, we ensure that no matter which widget has the focus the command will be invoked by the keyboard shortcut.

Events

Of course, the possible events (or sequences of events) vary from widget to widget. In addition, these events can be specified in a few ways.

The example below uses two types of events. A single key press event, such as "C" or "O," can invoke a command and is specified by just its character, whereas the event of pressing the return key is specified using Return. In the following, we bind the key presses to the top-level window and the return event to any button with the default class TButton.

```

window <- tkoplevel()
label <- ttklabel(window, text = "Click Ok for a message")
button1 <- ttkbutton(window, text = "Cancel",
                    command = function() tkdestroy(window))
button2 <- ttkbutton(window, text = "Ok", command=function() {
  print("initiate an action")
})
sapply(list(label, button1, button2), tkpack)
##
tkbind(window, "C", function() tcl(button1, "invoke"))
tkconfigure(button1, underline = 0)

```



Figure 17.4: Simple GUI showing buttons with underline property. The underlined letters match bindings to the top-level window to invoke the button.

```
##
tkbind(window, "O", function() tcl(button1, "invoke"))
tkconfigure(button2, underline = 0)
tkfocus(button2)
##
tkbind("TButton", "<Return>", function(W) {
    tcl(W, "invoke")
})
```

We modified our buttons using the underline option to give the user an indication that the “C” and “O” keys will initiate some action (Figure 17.4). Our callbacks simply cause the appropriate button to invoke their command. The latter one uses a percent substitution (below), which is how Tk passes along information about the event to the callback.

Events with modifiers More complicated events can be described with the pattern

`<modifier-modifier-type-detail>`.

Examples of a “type” are `<KeyPress>` or `<ButtonPress>`. The event `<Control-q>`, used above, has the type `q` and modifier `Control`, whereas `<Double-Button-1>` uses the detail `1` to indicate which mouse button. The full list of modifiers and types is described in the manual page for `bind`. Some familiar modifiers are `Control`, `Alt`, `Double`, and `Triple`. The event types are the standard X event types, along with some abbreviations. These are also listed in the `bind` manual page. Some commonly used ones are `Return` (as above), `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 as `<<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 function takes at least two arguments, an event name and a sequence which 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 `button`:

```
tkevent.generate(button, "<<Save>>")
```

Example 17.1 uses virtual events to implement drag and drop features.

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. The basic callback function need not have any arguments and those that do have only percent substitutions passed in.

The callback’s return value is generally not important, although we shall see that within the validation framework of entry widgets (Section 19.2) it can matter.⁵

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 top-level windows can differ.) Calling

⁵The difference in processing of return values can make porting some `Tk` code to `tcltk` difficult. For example, the `break` command to stop a chain of callbacks does not work.

tkbind another time will replace the callback. To remove a callback, simply assign a new callback that does nothing.⁶

Percent substitutions

We cannot pass arbitrary user data to a callback; rather, such values must be found through R's usual scoping rules. However, 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 manual page. Some useful ones are x and X to specify the relative or absolute x-postion 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.

The following trivial example illustrates the steps, whereas Example 17.1 will put these to use.

```
window <- tktoplevel()
button <-
  ttkbutton(window, text = "Click me for the x,y position")
tkpack(button)
tkbind(button, "<ButtonPress-1>", function(W, x, y, X, Y) {
  print(W)                                # an ID
  print(c(x, X))                          # character class
  print(c(y, Y))
})
```

The after command The Tcl command after will execute a command after a certain delay (specified in milliseconds as an integer) while not interrupting the control flow while it waits for its delay. The function is called in a manner like this:

```
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:

```
after_ID <- ""
some_flag <- TRUE
```

⁶This event handling can prevent us from being able to port some Tk code into tcltk. In those cases, we can consider sourcing in Tcl code directly.

```

repeat_call <- function(ms = 100, f) {
  after_ID <- tcl("after", ms, function() {
    if(someFlag) {
      f()
      after_ID <- repeat_call(ms, f)
    } else {
      tcl("after", "cancel", after_ID)
    }
  })
}
repeat_call(2000, function() {
  print("Running. Set someFlag <- FALSE to stop.")
})

```

Example 17.1: Drag-and-drop

This relatively involved example⁷ shows several different uses of the event framework to implement drag and drop behavior between two widgets. It certainly can be skipped on first reading.

In `tcltk` much more work is involved with drag and drop, than with `RGtk2` and `qtbase`, as there is no provided framework.

Here we go through the steps needed to make one widget a drop source and the other 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 listening to these events, then the drag and drop can be processed.

To begin, we create a simple GUI to hold three widgets. We use buttons for drag and drop, but only for convenience. Other widgets would be used in a real application.

```

window <- tkoplevel()
b_drag <- ttkbutton(window, text = "Drag me")
b_drop <- ttkbutton(window, text = "Drop here")
tkpack(b_drag)
tkpack(ttklabel(window, text = "Drag over me"))
tkpack(b_drop)

```

Before beginning, we define three global variables that can be shared among drop sources to keep track of the drag and drop state.

```

.dragging <- FALSE           # currently dragging?
.drag_value <- ""           # value to transfer
.last_widget_id <- ""       # last widget dragged over

```

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.

⁷The idea for the example code originated with <http://wiki.tcl.tk/416>

```
tkbind(b_drag, "<ButtonPress-1>", function(W) {
    .dragging <<- TRUE
    .drag_value <<- as.character(tkcget(W, text = NULL))
    .last_widget_id <<- as.character(W)
})
```

This initiates the dragging immediately. A more common strategy is to record the position of the mouse click and then initiate the dragging after a certain minimal movement is detected.

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 comment refers to a web page showing how we can put in a custom cursor from an xbm file, but this doesn't work for all platforms (e.g., OS X and Aqua). After setting the cursor, we find the ID of the widget the cursor is hovering over. We use `tkwinfo` to find the widget containing the x and 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(window, "<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

    win <- tkwinfo("containing", X, Y) # widget mouse is over
    if(as.logical(tkwinfo("exists", win))) # does widget exist?
        tkevent.generate(win, "<<DragOver>>")

    ## generate drag leave if we left last widget
    if(as.logical(tkwinfo("exists", win)) &&
        nchar(as.character(win)) > 0 &&
        length(.last_widget_id) > 0) { # if not character(0)
        if(as.character(win) != .last_widget_id)
            tkevent.generate(.last_widget_id, "<<DragLeave>>")
    }
    .last_widget_id <<- as.character(win)
})
```

Finally, if the button is released, we generate the `<<DragLeave>>` and, most importantly, `<<DragDrop>>` virtual events for the widget we are over.

```
tkbind(b_drag, "<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
.last_widget_id <- ""
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(b_drop, "<<DragOver>>", function(W) {
  if(.dragging)
    tcl(W, "state", "active")
})

```

If the drag event leaves the widget without dropping, we change the state back to not active.

```

tkbind(b_drop, "<<DragLeave>>", function(W) {
  if(.dragging) {
    tkconfigure(W, cursor = "")
    tcl(W, "state", "!active")
  }
})

```

Finally, if the <<DragDrop>> virtual event occurs, we set the widget value to that stored in the global variable `.drag_value`.

```

tkbind(b_drop, "<<DragDrop>>", function(W) {
  tkconfigure(W, text = .drag_value)
  .drag_value <- ""
})

```


This page intentionally left blank

Tcl/Tk: Layout and Containers

18.1 Top-level windows

Top-level windows are created through the `tktoplevel` constructor. Basic options include the ability to specify the preferred width and height and to specify a menu bar through the `menu` argument. (Menus will be covered in Section 20.3.)

Other properties can be queried and set through the Tk command `wm`. This command has several subcommands, leading to `tcltk` functions with names such as `tkwm.title`, the function used to set the window title. For 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.

Suppressing the initial drawing When a top-level window is constructed there is no option for it not to be shown. However, we can use the `tclServiceMode` function to suspend/resume drawing of any widget through Tk. This function takes a logical value indicating that the updating of widgets should be suspended. We can set the value to `FALSE`, initiate the widgets, then set to `TRUE` to display the widgets. To iconify an already drawn window can be done through the `tkwm.withdraw` function and reversed with the `tkwm.deiconify` function. Either of these can be useful 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` arguments of the constructor. Negative values mean the window will not request any size. 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, as was described for `tkwininfo` in Section 17.3. If this string is empty, then the window will resize to accommodate its child components.

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 top-level window `window`.

```
tkwm.resizable(w, FALSE, FALSE) # width first
```

When a window is resized, we 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 resizable windows, the `ttksizegrip` widget can be used to add a visual area (usually the lower-right corner) for the user to grab on to with a mouse for resizing the window. On some operating systems (e.g., Mac OS X) these are added automatically by the window manager.

Dialog windows For dialogs, a top-level window can be related to a different top-level window. The function `tkwm.transient` allows one to specify the master window as its second argument (cf. Example 18.1). The new window will mirror the state of the master window, including the case when the master is withdrawn.

For some dialogs it may be desirable not to have the window manager decorate the window with a title bar, etc. The command `tkoplevel wm overriddenirect logical` takes a logical value indicating whether the window should be decorated. 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 be specified only once for the top-level window (cf. Section 17.4 where keyboard shortcuts are defined this way).

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 window 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 `window` is a top-level window, and `entry` a text entry widget (cf. `tktext` in Section 20.2), then the following snippet of code would check to see whether the text widget has been modified before closing the window. This uses a modal message box described in Section 19.1.

```
tkwm.protocol(window, "WM_DELETE_WINDOW", function() {
    modified <- tcl(entry, "edit", "modified")
    if(as.logical(modified)) {
```

```

response <-
  tkmessageBox(icon = "question",
              message = "Really close?",
              detail = "Changes need to be saved",
              type = "yesno",
              parent = window)
if(as.character(response) == "no")
  return()
}
tkdestroy(window)           # otherwise close
})

```

Example 18.1: A window constructor

This example shows a possible constructor for top-level windows allowing some useful options to be passed in. We use the upcoming `ttkframe` constructor and `tkpack` command.

```

newWindow <- function(title, command, parent,
                    width, height) {
  window <- tktoplevel()

  if(!missing(title)) tkwm.title(window, title)

  if(!missing(command))
    tkwm.protocol(window, "WM_DELETE_WINDOW", function() {
      if(command())           # command returns logical
        tkdestroy(window)
    })

  if(!missing(parent)) {
    parent_window <- tkwinfo("top-level", parent)
    if(as.logical(tkwinfo("viewable", parent_window))) {
      tkwm.transient(window, parent)
    }
  }

  if(!missing(width)) tkconfigure(window, width = width)
  if(!missing(height)) tkconfigure(window, height = height)

  window_system <- tclvalue(tcl("tk", "windowingsystem"))
  if(window_system == "aqua") {
    frame <- ttkframe(window, padding = c(3,3,12,12))
  } else {
    int_frame <- ttkframe(window, padding = 0)
    tkpack(int_frame, expand = TRUE, fill = "both")
    frame <- ttkframe(int_frame, padding = c(3,3,12,0))
    sizegrip <- ttksizegrip(int_frame)
  }
}

```

```
    tkpack(sizegrip, side = "bottom", anchor = "se")
  }
  tkpack(frame, expand = TRUE, fill = "both", side = "top")

  return(frame)
}
```

18.2 Frames

The `tkframe` constructor produces a themeable container that can be used to organize visible components within a GUI. As mentioned, for theme reasons, it is often the first thing packed within a top-level window.

The options include `width` and `height` to set the requested size. The `padding` option can be used to put space between the border and subsequent children. Frames can be decorated. Use the option `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"`, and `"sunken"`.

Label frames

The `tklabelframe` 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`, and `s`). The default is theme dependent, although typically `"nw"` (upper left).

Separators As an alternative to a border, the `tkseparator` widget can be used to place a single line to separate areas in a GUI. 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.

18.3 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 reallocate space to a GUI's components quickly when a window is resized. The `tkpack` function will place children into their parent in a boxlike manner. We have seen several examples in the text that use nested boxes to construct quite flexible layouts. Example 18.4

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.¹

A GUI may use a mix of `pack` and `grid` to manage the child components, but all immediate 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. There are many options to adjust this default behavior.

The `side` option can take a value of "left", "right", "top" (default), or "bottom" to adjust where the children are placed. Unlike GTK+ or Qt, where boxes are packed in just one direction, these can be mixed and matched, but sticking to just one direction is typical, with nested frames to give additional flexibility.

before, after The `before` and `after` 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.

forget Child components can be forgotten by the window manager, un-mapping them but not destroying them, with the `tkpack forget` subcommand, or the convenience function `tkpack.forget`. Example 20.5 shows a usage. After a child component is removed this way, it can be replaced in the GUI using a geometry manager.

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.

These commands are illustrated below, where we show how we might implement a ticker tape effect, where words scroll to the left.

```
window <- tkoplevel()
frame <- ttkframe(window, padding = c(3,3,12,12))
```

¹An excellent online reference, albeit for Perl/Tk, is *Learning Perl/Tk: Graphical User Interfaces with Perl*, by Nancy Walsh. See <http://www.rigacci.org/docs/biblio/online/lperltk/ch02.html> for information about this topic.

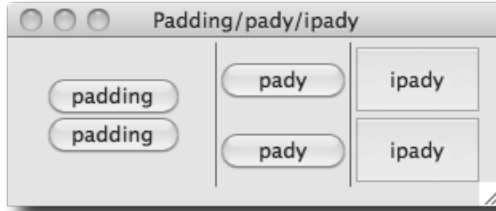


Figure 18.1: Various ways to put padding between widgets using `tkpack`. The `padding` option for the box container puts padding around the cavity for all the widgets. The `pady` option for `tkpack` puts padding around the top and bottom on the border of each widget. The `ipady` option for `tkpack` puts padding within the top and bottom of the border for each child (breaking the theme under Mac OS X).

```
tkpack(frame, expand = TRUE, fill = "both")
#
x <- strsplit("Lorem ipsum dolor sit amet ...", "\\s")[[1]]
labels <- lapply(x, function(i) ttklabel(frame, text = i))
sapply(labels, function(i) tkpack(i, side = "left"))
#
rotateLabel <- function() {
  children <- as.character(tkpack("slaves", frame))
  tkpack.forget(children[1])
  tkpack(children[1], after = children[length(children)],
          side = "left")
}
```

We could use the `after` command to do this in the background, but here we just rotate the values in a blocking loop:

```
for(i in 1:20) {rotateLabel(); Sys.sleep(1)}
```

Specifying space around the children 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 18.1 has an example. In the above options, the `x` and `y` indicate left-right space and 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, `padx` is for 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 we can easily add padding around all the children of the frame `frame` using the `tkpack` "configure" subcommand.

```
all_children <- as.character(tkwininfo("children", frame))
```

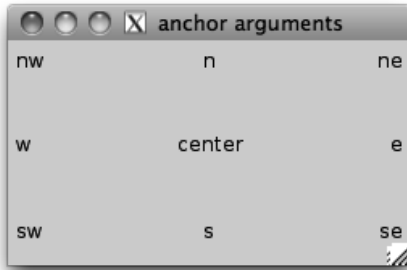


Figure 18.2: The anchor argument is specified through compass directions.

```
sapply(all_children, tkpack.configure, padx = 10, pady = 5)
```

Cavity model The packing algorithm, as described in the Tk documentation, is based on arranging where to place a slave into the rectangular unallocated space called a “cavity.” We use the nicer terms “child component” and “box” to describe this. 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.

By default, the child will be placed centered along the edge of the box within the allocated space with blank space, if any, on both sides.

The anchor, expand, fill arguments 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 (e.g. “`n`” or “`se`”), leaving blank space around the child (Figure 18.2).

An alternative is to have one or more of the widgets expand to fill the available space. Each child packed in with the option `expand` set to `TRUE` will have the extra space allocated to it in an even manner. 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 expand the child’s size in just one direction, the latter in both.

Example 18.2: Expand/fill options for `tkpack`

Figure 18.3 shows examples of different values for “`fill`” when `ex-`

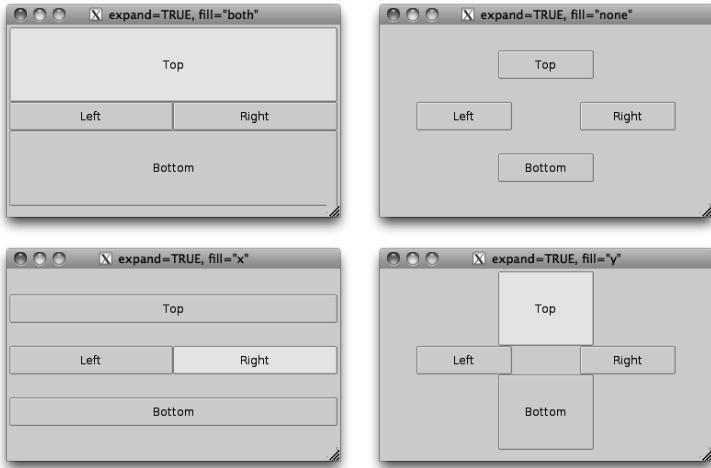


Figure 18.3: Similar layout with `expand=TRUE` but different values of `fill`. The space allocated to the top and bottom buttons through expansion fills the vertical area, as these were added with `side` set to `"top"` and `"bottom"`, respectively, whereas the left and right buttons expand in the horizontal direction, as they were added with `sides` `"left"` and `"right"`. The different `fill` values direct the buttons to take up this allocated space in different manners.

`pack=TRUE` is specified. Following an example of Walsh^[13] we used the following code to create the images:

```

window <- tkoplevel()
tkwm.title(window, "Expand/Fill arguments")
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both")
##
pack_btn <- function(txt, ...)
  tkpack(button <- ttkbutton(frame, text = txt), ...)
##
pack_btn("Top",    side="top",    expand=TRUE, fill="both")
pack_btn("Bottom", side="bottom", expand=TRUE, fill="both")
pack_btn("Left",   side="left",   expand=TRUE, fill="both")
pack_btn("Right",  side="right",  expand=TRUE, fill="both")

```

Modifying the fill styles was easy. For example,

```

children <- as.character(tkwininfo("children", frame))
sapply(children, tkpack.configure, fill = "none")

```

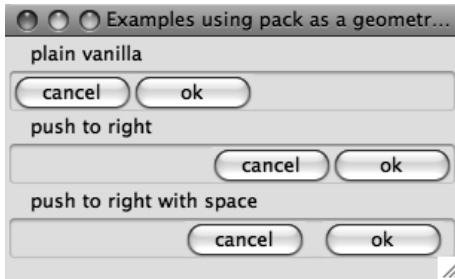


Figure 18.4: Demonstration of using `tkpack` options showing effects of using the `side` and `padx` options to create dialog buttons.

Not enough space When the top-level window does not have sufficient space to satisfy the combined size requests of its child components, either some widgets will be covered or we can resize the top-level window. When components are covered, the ones that are packed in first are given highest priority in the size request.

To force a recomputation of the size of the top-level window, we can call the `wm geometry` subcommand with an empty string:

```
tkwm.geometry(window, "")
```

The top-level window, `window` above, can be recovered from a child component, say `button`, through:

```
tkwinfo("top-level", button)
```

propagate In Example 20.3 we define a convenience function for creating a table widget. There we have a call to the subcommand `pack propagate`. This prevents the querying of the child widgets to compute the size request. In the example, this is useful, as the scroll bars used should depend on the size requested by the parent and not the underlying table widget.

Example 18.3: 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 that the buttons are packed in left to right, not top to bottom.

```
frame_1 <- ttklabelframe(frame, text="plain vanilla")
tkpack(frame_1, expand = TRUE, fill = "x")
l <- function(f)
  list(ttkbutton(f, text="cancel"), ttkbutton(f, text="ok"))
sapply(l(frame_1), tkpack, side = "left")
```



Figure 18.5: Example of a simple dialog.

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.

```
frame_2 <- ttklabelframe(frame, text = "push to right")
tkpack(frame_2, expand = TRUE, fill = "x")
tkpack(ttklabel(frame_2, text = " "),
       expand = TRUE, fill = "x", side = "left")
sapply(l(frame_2), tkpack, side = "left")
```

Finally, we add some padding to conform to Apple's design specification that such buttons should have a 12-pixel separation.

```
frame_3 <- ttklabelframe(frame, text="push right with space")
tkpack(frame_3, expand = TRUE, fill = "x")
tkpack(ttklabel(frame_3, text = " "), expand=TRUE, fill="x",
       side = "left")
sapply(l(frame_3), tkpack, side = "left", padx = 6)
```

Example 18.4: 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 it's not written as a function, we set aside the values that would be passed in if it were.

```
title <- "message dialog"
message <- "Do you like tcltk so far?"
parent <- NULL
tkimage.create("photo", "::img::tclLogo",
              file = system.file("images", "tclp.gif",
                                package = "ProgGUIinR"))
```

The main top-level window is given a title then withdrawn while the GUI is created.

```
window <- tktoplevel()
tkwm.title(window, title)
tkwm.state(window, "withdrawn")
frame <- ttkframe(window, padding = c(3, 3, 12, 12))
tkpack(frame, expand = TRUE, fill = "both")
```

As usual, we added a frame so that any themes are respected.

If the parent is non-null and 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)) {
  parent_window <- tkwinfo("toplevel", parent)
  if(as.logical(tkwinfo("viewable", parent_window))) {
    tkwm.transient(window, parent)
    ## have fun with OS X
    if(as.character(tcl("tk", "windowingsystem")) == "aqua") {
      tcl("wm", "attributes", parent_window, notify = TRUE)
      tclbind(parent_window, "<Enter>", function()
        tcl("wm", "attributes", parent_window,
            notify = FALSE))      # stop bounce
    }
  }
}
```

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,

```
label <- ttklabel(frame, image = "::img::tclLogo", padding=5)
tkpack(label, side = "left")
```

A nested frame will be used to lay out the message area and button area. Since the `tkpack` default is to pack in top to bottom, no side specification is made.

```
frame_1 <- ttkframe(frame)
tkpack(frame_1, expand = TRUE, fill = "both")
#
m <- ttklabel(frame_1, text = message)
tkpack(m, expand = TRUE, fill = "both")
```

The buttons have their own frame, as they are laid out horizontally.

```
frame_2 <- ttkframe(frame_1)
tkpack(frame_2)
```

The callback function for the "OK" button prints a message then destroys the window.

```
ok_callback <- function() {
  print("That's great")
  tkdestroy(window)
}
ok_button <- ttkbutton(frame_2, text = "OK",
  command = ok_callback)
```

```
cancel_button <- ttkbutton(frame_2, text = "Cancel",
                           command = function() tkdestroy(window))
#
tkpack(ok_button, side = "left", padx = 12) # give some space
tkpack(cancel_button)
```

As our interactive behavior is consistent for both buttons, we make a binding to the `TButton` class, not to each button individually. The first will invoke the button command when the return key is pressed; the latter two will highlight a button when the focus is on it.

```
tkbind("TButton", "<Return>", function(W) tcl(W, "invoke"))
tkbind("TButton", "<FocusIn>", function(W)
      tcl(W, "state", "active"))
tkbind("TButton", "<FocusOut>", function(W)
      tcl(W, "state", "!active"))
```

Now we bring the dialog back from its withdrawn state, fix the size, and set the initial focus on the “OK” button.

```
tkwm.state(window, "normal")
tkwm.resizable(window, FALSE, FALSE)
tkfocus(ok_button)
```

Grid

The `tkgrid` geometry manager is used to align 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 . If desired, 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` and `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. Allocation is based on proportions, not specified sizes. The 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 twice as much extra space is allocated for the second row. The specific row or column must also be specified. Again, rows and columns are referenced starting with 0, not the

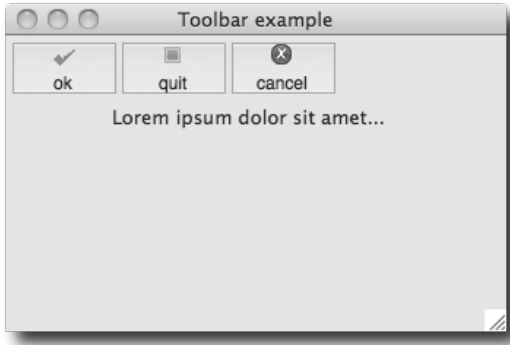


Figure 18.6: Illustration of using `tkpack` and `tkgrid` to make a toolbar.

usual R-like 1. 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 The `tkpack` command had options `anchor`, `expand` and `fill` to control what happens when more space is available than requested by a child component. The `sticky` option for `tkgrid` combines these. 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 directions, like `expand=TRUE`, `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 18.5: 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.

```
window <- tkoplevel(); tkwm.title(window, "Toolbar example")
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both")
```

Our example has two main containers: one to hold the toolbar buttons and one to hold the main content.

```
tool_bar_frame <- ttkframe(frame, padding = 0)
content_frame <- ttkframe(frame, 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 vertically if the window does.

```
tkgrid(tool_bar_frame, row = 0, column = 0, sticky = "we")
tkgrid(content_frame, row = 1, column = 0, sticky = "news")
tkgrid.rowconfigure(frame, 0, weight = 0)
tkgrid.rowconfigure(frame, 1, weight = 1)
tkgrid.columnconfigure(frame, 0, weight = 1)
#
txt <- "Lorem ipsum dolor sit amet..." # sample text
tkpack(ttklabel(content_frame, text = txt))
```

Now to add some buttons to the toolbar. We first show how to create a new style for a button (`Toolbar.TButton`), slightly modifying the themed button to set the font and padding, and eliminate the border if the operating system allows.

```
tcl("ttk::style", "configure", "Toolbar.TButton",
    font = "helvetica 12", padding = 0, borderwidth = 0)
```

This `make_icon` function finds stock icons from the `gWidgets` package and adds them to a button.

```
make_icon <- function(parent, stock_name, command = NULL) {
  icon_file <- system.file("images",
                           paste(stock_name, "gif", sep = "."),
                           package = "gWidgets")
  if(nchar(icon_file) == 0) {
    b <- ttkbutton(parent, text = stock_name, width = 6)
  } else {
    icon_name <- paste("::img::", stock_name, sep = "")
    tkimage.create("photo", icon_name, file = icon_file)
    b <- ttkbutton(parent, image = icon_name,
                  style = "Toolbar.TButton", text=stock_name,
                  compound = "top", width = 6)
    if(!is.null(command))
      tkconfigure(b, command = command)
  }
  return(b)
}
```

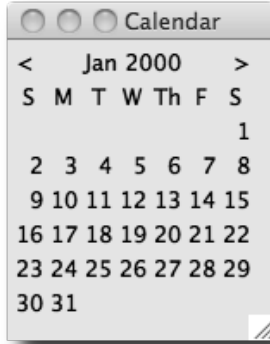


Figure 18.7: A monthly calendar illustrating various layouts.

To illustrate, we pack in some icons. Here we use `tkpack`. We do not use `tkpack` and `tkgrid` to manage children of the same parent, but these are children of `tool_bar_frame`, not `frame`.

```
sapply(c("ok", "quit", "cancel"), function(i)
  tkpack(make_icon(tool_bar_frame, i), side = "left"))
```

These two bindings change the state of the buttons when the mouse hovers over one of them:

```
setState <- function(W, state) tcl(W, "state", state)
tkbind("TButton", "<Enter>", function(W) setState(W, "active"))
tkbind("TButton", "<Leave>", function(W) setState(W, "!active"))
```

If one wished to restrict the above to just the toolbar buttons, one could check for the style of the button, as with:

```
function(W) {
  if(as.character(tkcget(W, "-style")) == "Toolbar.TButton")
    cat("... do something for toolbar buttons ...")
}
```

Example 18.6: Using `tkgrid` to lay out a calendar

This example shows how to create a simple calendar using a grid layout. (No such widget is standard with `tcltk`.) We use some data functions for the `ProgGUIinR` package. The actual use of `tkgrid` is straightforward once the appropriate row and column are figured out.

```
make_month <- function(parent, year, month) {
  ## add headers
  days <- c("S", "M", "T", "W", "Th", "F", "S")
  sapply(1:7, function(i) {
```



```
    label <- ttklabel(parent, text = days[i])
    tkgrid(label, row = 0, column = i-1, sticky = "")
  })
  ## add days
  sapply(seq_len(ProgGUIinR::days.in.month(year, month)),
        function(day) {
          label <- ttklabel(parent, text = day)
          row <- ProgGUIinR::week.of.month(year, month, day)
          col <- ProgGUIinR::day.of.week(year, month, day)
          tkgrid(label, row = 1 + row, column = col,
                sticky = "e")
        })
}
```

Next, we would like to incorporate the calendar widget into an interface that allows the user to scroll through month-by-month beginning with:

```
year <- 2000; month <- 1
```

Our basic layout will use a box layout with a nested layout for the step-through controls and another holding the calendar widget.

```
window <- tktoplevel()
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both", side = "top")
c_frame <- ttkframe(frame)
cal_frame <- ttkframe(frame)
tkpack(c_frame, fill = "x", side = "top")
tkpack(cal_frame, expand = TRUE, anchor = "n")
```

Our step-through controls are packed in through a horizontal layout. We use anchoring and `expand=TRUE` to keep the arrows on the edge and the label with the current month centered, should the container be resized.

```
previous_button <- ttklabel(c_frame, text = "<")
next_button <- ttklabel(c_frame, text = ">")
current_month <- ttklabel(c_frame)
#
tkpack(previous_button, side = "left", anchor = "w")
tkpack(current_month, side = "left", anchor = "center",
       expand = TRUE)
tkpack(next_button, side = "left", anchor = "e")
```

The `set_month` function first removes the previous calendar container and then redefines one to hold the monthly calendar. Then it adds in a new monthly calendar to match the year and month. The call to `make_month` creates the calendar. Packing in the frame after adding its child components makes the GUI seem much more responsive.

```
set_month <- function() {
```

```

tkpack("forget", cal_frame)
cal_frame <- ttkframe(frame)
make_month(cal_frame, year, month)
tkconfigure(current_month,           # month label
             text = sprintf("%s %s", month.abb[month], year))
tkpack(cal_frame)
}
set_month()                          # initial calendar

```

The arrow labels are used to scroll, so we connect to the Button-1 event the corresponding commands. This shows the binding to decrement the month and year using the global variables `month` and `year`.

```

tkbind(previous_button, "<Button-1>", function() {
  if(month > 1) {
    month <- month - 1
  } else {
    month <- 12; year <- year - 1
  }
  set_month()
})

```

Our calendar is static, but if we wanted to add interactivity to a mouse click, we could make a binding as follows:

```

tkbind("TLabel", "<Button-1>", function(W) {
  day <- as.numeric(tkget(W, "-text"))
  if(!is.na(day))
    print(sprintf("You selected: %s/%s/%s", month, day, year))
})

```

18.4 Other containers

Tk provides just a few other basic containers. Here we describe paned windows and notebooks.

Paned windows

A paned window, with sashes to control the individual pane sizes, 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 of how the sash is drawn.

The returned object can be used as a parent container, although we do not use the geometry managers to manage these objects. Instead, the `add` command is used to add a child component. For example:

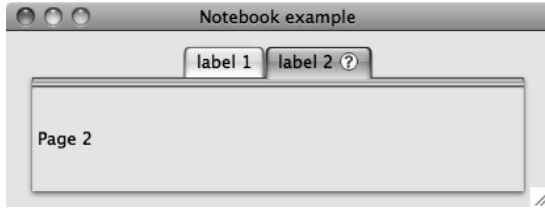


Figure 18.8: A basic notebook under Mac OS X.

```

window <- tktoplevel()
tkwm.title(window, "Paned window example")
paned <- ttkpanedwindow(window, orient = "horizontal")
tkpack(paned, expand = TRUE, fill = "both")
left <- ttklabel(paned, text = "left")
right <- ttklabel(paned, text = "right")
#
tkadd(paned, left, weight = 1)
tkadd(paned, right, weight = 2)

```

When resizing, the allocation of space among the children is determined by their associated `weight`, specified as an integer. The default uses even weights. Unlike with 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(paned, "forget", right)
tkadd(paned, right, weight = 2) ## how to add back

```

Sash position The sash between two children can be adjusted through the subcommand `ttkpanedwindow sashpos`. The index of the sash needs to be specified, 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 and used as follows:

```

width <- as.integer(tkwininfo("width", paned)) # or "height"
tcl(paned, "sashpos", 0, floor(0.75*width))

```

```
<Tcl> 54
```

Notebooks

Tabbed notebook containers are produced by the `ttknotebook` constructor. 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. Typically, the child components would be containers to hold more complicated layouts. 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.

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, as either 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 `notebook` is a `ttknotebook` object, then these commands would add pages (cf. Figure 18.8):

```
icon_file <- system.file("images",paste("help","gif",sep="."),
                        package = "gWidgets")
icon_name <- "::tcl::helpIcon"
tkimage.create("photo", icon_name, file = icon_file)
#
page2_label <- ttklabel(notebook, text = "Page 2")
tkadd(notebook, page2_label, sticky = "nsw", text="label 2",
      image = icon_name, compound = "right")
## put page 1 label first (a tabID of 0); use tkinsert
page1_label <- ttklabel(notebook, text = "Page 1")
tkinsert(notebook, 0, page1_label, sticky = "nsw",
        text = "label 1")
```

There are several useful subcommands for extracting 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. (There is no means to place close icons on the tabs.) Except for `tabs`, these require a specification of a tab ID.

```
tcl(notebook, "index", "current") # current page for tabID
```

```
<Tcl> 1
```

```
length(as.character(tcl(notebook,"tabs"))) # number of pages
```

```
[1] 2
```

```
tcl(notebook, "select", 0)           # select by index
tcl(notebook, "forget", page1_label) # "forget" removes a page
tcl(notebook, "add", page1_label)   # can be managed again.
```

The notebook state can be manipulated through the keyboard, provided traversal is enabled. This can be done through

```
tcl("ttk::notebook::enableTraversal", notebook)
```

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 icons, as in Qt.

Tcl/Tk: Dialogs and Widgets

This chapter covers both the standard dialogs provided by Tk and the various controls used to create custom dialogs. We begin with a discussion of the standard dialogs, then cover the basic controls in this chapter, leaving the next chapter for the more involved `tktext`, `ttktreeview`, and `tkcanvas` widgets.

19.1 Dialogs

Modal dialogs

The `tkmessageBox` constructor can be used to create simple modal dialogs, allowing a user to confirm an action. These use the native toolkit if possible. This constructor replaces the older `tkdialog` dialogs. 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 message dialog has an `icon` argument. The default icon is "info" but could also be "error", "question", or "warning". The buttons used are specified through 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:

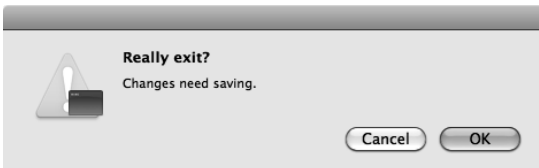


Figure 19.1: A basic modal dialog constructed by `tkmessageBox`.

```
tkmessageBox(title = "Confirm", message = "Really exit?",
             detail = "Changes need saving.",
             icon = "question", type = "okcancel")
```

The tkwait function If the default modal dialog is not enough – for instance there is no means to gather user input – then a top-level window can be made modal. The `tkwait` function will cause a top-level window to be modal, and `tkgrab.release` will return the interactivity for the window. We illustrate a simple use by example, beginning by adding a label to a window:

```
msg <- "We care ..."
dialog <- tkoplevel(); tkwm.withdraw(dialog)
tkwm.overrideRedirect(dialog, TRUE) # no decoration
frame <- ttkframe(dialog, padding = 5)
tkpack(frame, expand = TRUE, fill = "both")
tkpack(ttklabel(frame, text = msg), pady = 5)
```

There are different ways to use `tkwait`. The function `tkwait.window` will make a top-level window modal, waiting until it is destroyed. In the following we will use `tkwait.variable`. This will keep the window modal until a change to a specified variable, in this case `flag`. In the button's command we release the window then change this value, ending the wait.

```
flag <- tclVar("")
tkpack(ttkbutton(frame, text="dismiss", command=function() {
  tkgrab.release(dialog)
  tclvalue(flag) <- "Destroy"
}))
```

Now we show the window and wait for the `flag` variable to change.

```
tkwm.deiconify(dialog)
tkwait.variable(flag)
```

When the value of `flag` is changed in the callback the flow returns to the program.

File and directory selection

Tk provides constructors for selecting a file, selecting a directory or specifying a filename when saving. These are implemented by `tkgetOpenFile`, `tkchooseDirectory`, and `tkgetSaveFile`, respectively. Each of these can be called with no argument, and each returns a `TclObj` object. The value is empty when there is no selection made.

The dialog will appear in a relationship with a top-level window if the argument `parent` is specified. The `initialdir` and `initialfile` can

be used to specify the initial values in the dialog. The `defaultextension` argument can be used to specify a default extension for the file.

It can be convenient to filter the available file types that can be selected, when browsing for files. 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, we could use:

```
tkgetOpenFile(filetypes = paste(
    "{{jpeg files} {.jpg .jpeg} }",
    "{{png files} {.png}}",
    "{{All files} {*}}", sep = " ") # needs space
```

Extending this pattern is hopefully clear from above.

Example 19.1: A 'File' menu

To illustrate, a simple example for a file menu (Section 20.3) could include:

```
window <- tktoplevel()
tkwm.title(window, "File menu example")
menu_bar <- tkmenu(window)
tkconfigure(window, menu = menu_bar)
file_menu <- tkmenu(menu_bar)
tkadd(menu_bar, "cascade", label="File", menu = file_menu)
tkadd(file_menu, "command", label = "Source file...",
      command = function() {
        file_name <- tkgetOpenFile(filetypes=
            "{{R files} {.R}} {{All files} *}")
        if(file.exists(file_name <- as.character(file_name)))
            source(tclvalue(file_name))
      })
tkadd(file_menu, "command", label = "Save workspace as...",
      command = function() {
        file_name <- tkgetSaveFile(defaultextension = "Rsave")
        if(nchar(fname <- as.character(file_name)))
            save.image(file = file_name)
      })
tkadd(file_menu, "command", label="Set working directory...",
      command = function() {
        dir_name <- tkchooseDirectory()
        if(nchar(dir_name <- as.character(dir_name)))
            setwd(dir_name)
      })
```

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: `initial-`

color 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 quantities. There is no constructor in tcltk, but we can use the dialog as follows:

```
window <- tkoplevel()
tkwm.title(window, "Select a color")
frame <- ttkframe(window, padding = c(3,3,3,12))
tkpack(frame, expand = TRUE, fill = "both")
color_well <- tkcanvas(frame, width = 40, height = 16,
                      background = "#ee11aa",
                      highlightbackground = "#ababab")
tkpack(color_well)
tkpack(ttklabel(frame, text = "Click color to change"))
#
tkbind(color_well, "<Button-1>", function(W) {
  color <- tcl("tk_chooseColor", parent = W,
              title = "Set box color")
  color <- tclvalue(color)
  print(color)
  if(nchar(color))
    tkconfigure(W, background = color)
})
```

19.2 Selection widgets

This section covers the many 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.

Check buttons

The `ttkcheckbutton` constructor returns a `checkbutton` object. The check button's value (TRUE or FALSE) is linked to a Tcl variable which can be specified using a logical value. The check button label can also be specified through a Tcl variable using the `textvariable` option. Alternately, as

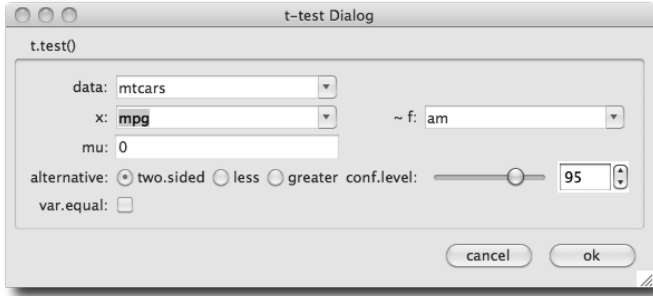


Figure 19.2: A dialog to collect values for a t -test (cf. Example 19.4) showing several of the selection widgets discussed in the section: a check button, radio button, combo boxes, a scale widget, and a spin box.

with the `ttklabel` constructor, the label can be specified through the `text` option. As well, we can specify an image 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 `frame` is a frame, we can create a new check button with the following:

```
value_var <- tclVar(TRUE)
callback <- function() print(tclvalue(value_var)) # uses global
label_var <- tclVar("check button label")
check_button <-
  ttkcheckbutton(frame, variable = value_var,
                 textvariable = label_var, command = callback)
tkpack(check_button)
```

A toggle button By default the widget draws with a checkbox. Optionally, the widget can be drawn as a button, which indicates a `TRUE` state by appearing depressed. This is done by using the style `Toolbutton`, as in:

```
tkconfigure(check_button, style = "Toolbutton")
```

In general, the “`Toolbutton`” style is for placing widgets into toolbars.

Avoiding global variables To avoid using a global variable here is not trivial. 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

percent substitution values. The variable holding the value can be found through:

```
tkcget(check_button, "variable" = NULL)
```

```
<Tcl> ::RTcl5
```

But then, we need a means to look up the variable from this id. Here is a wrapper for the `tclVar` function and a look up function that use an environment created by the `tcltk` package in place of a global variable.

```
our_tcl_var <- function(...) {  
  var <- tclVar(...)  
  .TkRoot$env[[as.character(var)]] <- var  
  var  
}  
## lookup function  
get_tcl_var_by_id <- function(id) {  
  .TkRoot$env[[as.character(id)]]  
}
```

Assuming we used `our_tcl_var` above, then the callback could be defined to avoid a (new) global variable by:

```
callback <- function(W) {  
  id <- tkcget(W, "variable" = NULL)  
  print(get_tcl_var_by_id(id))  
}
```

In Section 19.2 we demonstrate a better way – encapsulating the widget and its variable in a reference class so that we need not worry about scoping rules to reference the variable.

Radio buttons

Radio buttons are basically differently styled check buttons linked through a shared Tcl variable. Each radio button is constructed through the `ttk-radiobutton` constructor. Each button has both a value and a text label, which need not be the same. The `variable` option refers to the value. As with the `ttklabel` widget, the radio button labels can 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 can be used for selection of an alternative hypothesis, assuming `frame` 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) {
  radio_button <- ttkradiobutton(frame, variable = var,
                                text = i, value = i,
                                command = callback)
  tkpack(radio_button, side = "top", anchor = "w")
})
```

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.

```
txt_var <- tclVar("initial value")
entry <- ttkentry(window, textvariable = txt_var)
tkpack(entry)
```

We can get and set values using the Tcl variable.

```
tclvalue(txt_var)
```

```
[1] "initial value"
```

```
tclvalue(txt_var) <- "set value"
```

The `get` command can also be used.

```
tkget(entry)
```

```
<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 ($@x$), or one of the values "end" or "insert" to refer to the end of the current text and the insert point 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(entry, "end", "new text")
```

Or, we can delete a range of text, in this case the first four characters, using `tkentry delete`:

```
tkdelete(entry, 0, 4)
```

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.

The `tkentry icursor` command can be used to set the cursor position to the specified index.

```
tkicursor(entry, 0) # move to beginning
```

Finally, we note that the selection can be adjusted using the `tkentry selection range` subcommand. This takes two indices. Like `delete`, the first index specifies the first character of the selection, and the second indicates the character to the right of the selection boundary. The following example would select all the text.

```
tkselection.range(entry, 0, "end")
```

The `tkentry selection clear` subcommand clears the selection and `tkentry selection present` signals if a selection is currently made.

Events Several useful events include `<KeyPress>` and `<KeyRelease>` for key presses and `<FocusIn>` and `<FocusOut>` for focus events.

Example 19.2: Putting in an initial message

In this example, we show how to augment the `tkentry` widget to allow the inclusion of an initial message to direct the user. As soon as the user focuses the entry area, say by clicking the mouse on it, the initial message clears and the user can type in a value.

We use an R reference class for our programming, as it allows us to encapsulate the entry widget, its Tcl variable and the initial message. The main properties we have are defined via

```
setOldClass(c("tkwin", "tclVar"))
TtkEntry <- setRefClass("TtkEntry",
  fields = list(
    entry = "tkwin",      # entry
    tcl_var = "tclVar",  # text variable
    init_msg = "character"
  ))
```

We need to indicate to the user that the initial message is not the current text. We do so with a style that simply sets the foreground (text) color to gray:

```
.Tcl("ttk::style configure Gray.TEntry -foreground gray")
```

Now we create methods to accommodate the initial message. We have methods `is_init_msg`, to compare the current text with the initial message, and `show_init_msg` and `hide_init_msg` to toggle the messages. The only novelty is using the `style` option for a themeable widget.

```
TtkEntry$methods(
    is_init_msg = function() {
        "Is the init text showing?"
        as.character(tclvalue(tcl_var)) == init_msg
    },
    hide_init_msg = function() {
        "Hide the initial text"
        if(is_init_msg()) {
            tkconfigure(entry, style = "TEntry")
            set_text("", hide = FALSE)
        }
    },
    show_init_msg = function() {
        "Show the initial text"
        tkconfigure(entry, style = "Gray.TEntry")
        set_text(init_msg, hide = FALSE)
    }
})
```

Our accessor methods, `set_text` and `get_text`, must work around a possible initial message.

```
TtkEntry$methods(
    set_text = function(text, hide = TRUE) {
        "Set text into widget"
        if(hide) hide_init_msg()
        tcl_var_local <- tcl_var # avoid warning
        tclvalue(tcl_var_local) <- text
    },
    get_text = function() {
        "Get the text value"
        if(!is_init_msg())
            as.character(tclvalue(tcl_var))
        else
            ""
    }
})
```

In the `initialize` method, we will add bindings to switch between the initial message and the entry area. We use the `focus in` and `out` events to initiate this.

```
TtkEntry$methods(
    add_bindings = function() {
        "Add focus bindings to make this work"
        tkbind(entry, "<FocusIn>", hide_init_msg)
    }
})
```

```
        tkbind(entry, "<FocusOut>", function() {
            if(nchar(get_text()) == 0)
                show_init_msg()
        })
    })
```

Our initialization method follows.

```
TtkEntry$methods(
  initialize = function(parent, text, init_msg = "", ...) {
    tcl_var <- tclVar()
    entry <- ttkentry(parent, textvariable = tcl_var)
    init_msg <- init_msg
    ##
    if(missing(text))
      show_init_msg()
    else
      set_text(text)
    add_bindings()
    callSuper(...)
  })
```

Finally, to use this widget, we call its new method to create an instance. The actual entry widget is kept in the `e` field, so we pack in `widget$entry`.

```
window <- tkoplevel()
widget <- TtkEntry$new(parent = window,
                      init_msg = "type value here")
tkpack(widget$entry)
#
button <- ttkbutton(window, text = "focus out onto this",
                    command = function() {
                        print(widget$get_text())
                    })
tkpack(button)
```

Example 19.3: Using validation for dates

As previously mentioned, there is no native calendar widget in `tcltk`. This example shows how we 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; 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 validation.

To begin, we define some patterns for acceptable date formats.

```
date_patterns <- c()
for(i in list(c("%m", "%d", "%Y"),          # U.S. style
              c("%m", "%d", "%y"))) {
  for(j in c("/", "-", " "))
    date_patterns[length(date_patterns)+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.

```
is_valid_date <- function(W, P) { # P is the current value
  for(i in date_patterns) {
    date <- try( as.Date(P, format = i), silent = TRUE)
    if(!inherits(date, "try-error") && !is.na(date)) {
      tkconfigure(W, foreground = "black") # or use 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.

```
indicate_invalid_date <- 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.

```
entry <- ttkentry(frame, validate = "focusout",
                 validatecommand = is_valid_date,
                 invalidcommand = indicate_invalid_date)
button <- ttkbutton(frame, text = "click") # focus target
sapply(list(entry, button), tkpack, side = "left", padx = 2)
```

Combo boxes

The `ttkcombobox` constructor returns a combo box object allowing for selection from a list of values, or, with the appropriate option, allowing the user to specify a value using an entry widget. The value of the combo box 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 us to coerce the results to the desired class.)

Unlike with GTK+ and Qt there is no option to include images in the displayed text. We can adjust the alignment through the `justify` options. By default, a user can add in additional values through the entry-widget part of the combo box. The `state` option controls this, with the default "normal" and the value "readonly" as an alternative. For editable combo boxes, the widget also supports some of the `ttkentry` commands just discussed.

To illustrate, again suppose `frame` is a parent container. Then we begin by defining some values to choose from and a Tcl variable.

```
values <- state.name
var <- tclVar(values[1]) # initial value
```

The constructor call is as follows:

```
combo_box <- ttkcombobox(frame,
                        values = values,
                        textvariable = var,
                        state = "normal", # or "readonly"
                        justify = "left")
tkpack(combo_box)
```

The possible values the user can select from can be configured after construction through the `values` option:

```
tkconfigure(combo_box, values = tolower(values))
```

There is one case where the above won't work: when there is a single value and this value contains spaces. In this case, we can coerce the value to be of class `tclObj`:

```
tkconfigure(combo_box, values = as.tclObj("New York"))
```

Setting the value Setting values can be done through the Tcl variable. As well, the value can be set by a matching value using the `ttkcombobox set` subcommand through `tkset` or by index (0-based) using the `ttkcombobox current` sub command.

```
tclvalue(var) <- values[2]           # using tcl variable
tkset(combo_box, values[4])         # by value
tcl(combo_box, "current", 4)        # by index
```

Getting the value We can retrieve the selected object in various ways: from the Tcl variable, or the `ttkcombobox get` subcommand can be used through `tkget`.

```
tclvalue(var)                       # TCL variable
```

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

```
tkget(combo_box)                   # get subcommand
```

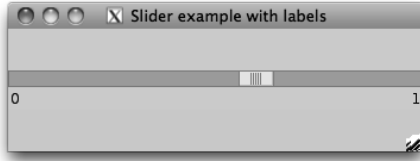
```
<Tcl> california
```

```
tcl(combo_box, "current")          # 0-based index
```

```
<Tcl> 4
```

Events The virtual event `<<ComboboxSelected>>` occurs with selection. When the combo box can be edited, a user may expect some action when the return key is pressed. This triggers a `<Return>` event. To bind to this event, we can do something like the following:

```
tkbind(combo_box, "<Return>", function(W) {
    val <- tkget(W)
    cat(as.character(val), "\n")
})
```

Figure 19.3: The `ttk::scale` widget with labels added.

Scale widgets

The `ttkscale` constructor to produce a themeable scale (slider) control is missing.¹ We can define our own simply enough:

```
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 `tkscale` (non-themed), has the option `resolution` to specify such a step amount. Also, the themeable slider does not have any label or tooltip indicating its current value.

As a work-around, we show how to display a vector of values by sliding through the indices and place labels at the ends of the slider to indicate the range (Figure 19.3). We write this using an R reference class.

```
Slider <-
  setRefClass("TtkSlider",
    fields = c("frame", "widget", "var", "x", "FUN"),
    methods = list(
      initialize = function(parent, x, ...) {
        initFields(x = x, var = tclVar(1),
                  FUN = NULL, frame = ttkframe(parent))
        widget <<- ttkscale(frame, from = 1, to = length(x),
                           variable = var, orient = "horizontal")
        ## For this widget, the callback is passed a value
        ## which we ignore here
        tkconfigure(widget, command = function(...) {
          if(is.function(FUN)) FUN(.self)
        })
        layout_gui()
        callSuper(...)
      },
```

¹As of the version of `tcltk` accompanying R 2.13.1.

```

layout_gui = function() {
  tkgrid(widget, row = 0, column = 0, columnspan = 3,
         sticky = "we")
  tkgrid(ttklabel(frame, text = x[1]),
         row = 1, column = 0)
  tkgrid(ttklabel(frame, text = x[length(x)]),
         row = 1, column = 2)
  tkgrid.columnconfigure(frame, 1, weight = 1)
},
add_callback = function(FUN) FUN <- FUN,
get_value = function() x[as.numeric(tclvalue(var))],
set_value = function(value) {
  "Set value. Value must be in x"
  ind <- match(value, x)
  if(!is.na(ind)) {
    var_local <- var
    tclvalue(var_local) <- ind
  }
}
))

```

To use this, we have:

```

window <- tktoplevel()
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both")
x <- seq(0,1,by = 0.05)
##
slider <- Slider$new(parent = window, x = x)
tkpack(slider$frame, expand = TRUE, fill = "x", anchor = "n")
##
slider$set_value(0.5)
print(slider$get_value())

```

```
[1] 0.5
```

As seen in the initialize and `get_value` methods, the `variable` option can be used for specifying a Tcl variable to record the value of the slider. This is convenient when the variable and widget are encapsulated into a class, as above. Otherwise, the `value` option is available. The `tkget` and `tkset` functions (using the `ttkscale` `get` and `ttkscale` `set` subcommands) can be used to get and set the value shown. They are used in the same manner as the same-named subcommands for a combo box.

The `add_callback` method can be used to add a callback function when the slider changes value.

```
slider$add_callback(function(obj) print(obj$get_value()))
```

We pass back in a reference to the object when we call this function, so there is no issue with finding the Tcl variable to get the value.

Spin boxes

A themeable spin box is introduced in Tk version 8.5.9. However, as of this writing, the libraries accompanying R for Windows are 8.5.8, so we will assume there is no themeable spin box widget. In Tk the `spinbox` command produces a non-themeable spin box. Again, there is no direct `tkspinbox` constructor, but one can be defined with:²

```
tkspinbox <- function(parent, ...)
  tkwidget(parent, "tk::spinbox", ...)
```

The non-themeable widgets have many more options than the themeable ones, as style properties can be set on a per-widget basis. We won't discuss those here. The spin box can be used to select from a sequence of numeric values or a vector of character values.

For example, the following allows a user to scroll in either direction through the fifty states of the U.S.

```
window <- tkoplevel()
spin_box <- tkspinbox(window, values = state.name, wrap=TRUE)
```

Whereas, this invocation will allow scrolling through a numeric sequence:

```
spin_box1 <- tkspinbox(window, from=1, to = 10, increment = 1)
```

The basic options to set the range for a numeric spin box are `from`, `to`, and `increment`. The `textvariable` option can be used to link the spin box to a Tcl variable. As usual, this allows the user to get and set the value displayed easily. Otherwise, the `tkget` and `tkset` functions can be used for these tasks.

As seen, in Tk spin boxes can also be used to select from a list of text values. These are specified through the `values` option. In the `state.name` example above, we set the `wrap` option to `TRUE`, so that the values wrap around when the end is reached.

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"). As with a combo box, when the Tk spin box displays character data and is in the "normal" state, the widget can be controlled like the entry widget of Section 19.2.

Example 19.4: 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 19.2.

²We augment this to compare the result of `tcl("info", "patchlevel")` to 8.5.9 and use "ttk::spinbox" when the underlying libraries support it.

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.

```
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)
```

This allows us to write a function to evaluate a *t*-test easily enough, although we don't illustrate that.

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.

```
label_frame <- ttklabelframe(frame, text = "t.test()",
                             padding = 10)
tkpack(label_frame, expand = TRUE, fill = "both",
        padx = 5, pady = 5)
```

The grid will have four columns, with columns 0 and 2 being for labels. We don't want the labels to expand the same way we want the widget columns to, so we assign different weights:

```
tkgrid.columnconfigure(label_frame, 0, weight = 1)
tkgrid.columnconfigure(label_frame, 1, weight = 10)
tkgrid.columnconfigure(label_frame, 2, weight = 1)
tkgrid.columnconfigure(label_frame, 3, weight = 10)
```

This helper function simplifies the task of adding a label.

```
put_label <- 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 combo box is used, although if a large number of data frames is a possibility, a different widget may be better suited. Also not shown are two similar calls to create combo boxes, `x_combo` and `factor_combo`, which allow the user to specify parts of a formula.

```
put_label(label_frame, "data:", 0, 0)
data_combo <- ttkcombobox(label_frame, state = "readonly",
                          values = ProgGUIinR:::avail_dfs(),
                          textvariable = e$data)
tkgrid(data_combo, row = 0, column = 1, sticky="ew", padx = 2)
tkfocus(data_combo) # give focus
```

We use a `ttkentry` widget for the user to specify a mean. For this purpose, the use is straightforward.

```
put_label(label_frame, "mu:", 2, 0)
mu_combo <- ttkentry(label_frame, textvariable = e$mu)
tkgrid(mu_combo, row = 2, column = 1, sticky = "ew", padx = 2)
```

The selection of an alternative hypothesis is a natural choice for a combo box or a radio-button group; we use the latter.

```
put_label(label_frame, "alternative:", 3, 0)
rb_frame <- ttkframe(label_frame)
sapply(c("two.sided", "less", "greater"), function(i) {
  radio_button <-
    ttkradiobutton(rb_frame, variable = e$alternative,
                  text = i, value = i)
  tkpack(radio_button, side = "left")
})
tkgrid(rb_frame, row = 3, column = 1, sticky = "ew", padx = 2)
```

Here we use a range widget to specify the confidence level. The slider is quicker to use but less precise than the spin box. By sharing a text variable, the widgets are automatically synchronized.

```
put_label(label_frame, "conf.level:", 3, 2)
conf_level_frame <- ttkframe(label_frame)
tkgrid(conf_level_frame, row = 3, column = 3, colspan = 2,
      sticky = "ew", padx = 2)
##
conf_level_scale <- ttkscale(conf_level_frame,
                          from = 75, to = 100,
                          variable = e$conf.level)
conf_level_spin <- tkspinbox(conf_level_frame,
                          from = 75, to = 100, increment = 1,
                          textvariable = e$conf.level, width = 5)
##
tkpack(conf_level_scale, expand = TRUE, fill = "y",
      side = "left")
tkpack(conf_level_spin, side = "left")
```

A checkbox is used to collect the logical value for `var.equal`:

```
put_label(label_frame, "var.equal:", 4, 0)
var_equal_check <-
  ttkcheckboxbutton(label_frame, variable = e$var.equal)
tkgrid(var_equal_check, row = 4, column = 1, stick = "w",
      padx = 2)
```

The dialog has standard "cancel" and "ok" buttons.

```
button_frame <- ttkframe(frame)
cancel_button <- ttkbutton(button_frame, text = "cancel")
ok_button <- ttkbutton(button_frame, text = "ok")
#
```

```
tkpack(button_frame, fill = "x", padx = 5, pady = 5)
tkpack(ttklabel(button_frame, text = " "), expand = TRUE,
       fill = "y", side = "left") # add a spring
sapply(list(cancel_button, ok_button), tkpack,
       side = "left", padx = 6)
```

For the `ok_button` button we want to gather the values and run the function. The `runTTest` function does this. We configure both buttons, then add to the default button bindings to invoke either of the buttons' commands when they have the focus and return is pressed.

```
tkconfigure(ok_button, command = runTTest)
tkconfigure(cancel_button,
            command = function() tkdestroy(window))
tkbind("TButton", "<Return>", function(W) tcl(W, "invoke"))
```

At this point our GUI is complete, but we would like to have it reflect any changes to the underlying R environment that affect its display. As such, we define a function, `update_ui`, which does two basic things: it searches for new data frames and it adjusts the controls depending on the current state. The work is cumbersome, as three different means are needed to disable the widgets.

```
update_ui <- function() {
  dfName <- tclvalue(e$data)
  curDfs <- ProgGUIinR:::avail_dfs()
  tkconfigure(data_combo, values = curDfs)
  if(!dfName %in% curDfs) {
    dfName <- ""
    tclvalue(e$data) <- ""
  }

  if(dfName == "") {
    ## 3 ways to disable needed!!
    x <- list(x_combo, factor_combo, mu_combo,
             conf_level_scale, var_equal_check, ok_button)
    sapply(x, function(W) tcl(W, "state", "disabled"))
    sapply(as.character(tkwininfo("children", rb_frame)),
           function(W) tcl(W, "state", "disabled"))
    tkconfigure(conf_level_spin, state = "disabled")
  } else {
    ## enable univariate, ok
    sapply(list(x_combo, mu_combo, conf_level_scale, ok_button),
           function(W) tcl(W, "state", "!disabled"))
    sapply(as.character(tkwininfo("children", rb_frame)),
           function(W) tcl(W, "state", "!disabled"))
    tkconfigure(conf_level_spin, state = "normal")

    DF <- get(dfName, envir = .GlobalEnv)
```



```
numVars <- get_numeric_vars(DF)
tkconfigure(x_combo, values = numVars)
if(! tclvalue(e$x) %in% numVars)
  tclvalue(e$x) <- ""

## bivariate
avail_factors <- get_two_level_factor(DF)
sapply(list(factor_combo, var_equal_check),
       function(W) {
         val <- if(length(avail_factors)) "!" else ""
         tcl(W, "state", sprintf("%sdisabled", val))
       })
tkconfigure(factor_combo, values = avail_factors)
if(!tclvalue(e$f) %in% avail_factors)
  tclvalue(e$f) <- ""

}
}
update_ui()
tkbind(data_combo, "<<ComboboxSelected>>", update_ui)
```

This function could be bound to a “refresh” button, or we could arrange to have it called in the background. Using the `after` command we could periodically check for new data frames, using a task callback. We can call this every time a new command is issued. As the call could potentially be costly, we call only if the available data frames have been changed. Here is one way to arrange that:

```
require(digest)
create_function <- function() {
  .DFs <- digest(ProgGUIinR:::avail_dfs())
  f <- function(...) {
    if((val <- digest(ProgGUIinR:::avail_dfs())) != .DFs) {
      .DFs <- val
      update_ui()
    }
  }
  return(TRUE)
}
}
```

Then to create a task callback we have:

```
id <- addTaskCallback(create_function())
```

Tcl/Tk: Text, Tree, and Canvas Widgets

This chapter focuses on a few of the more complex widgets of Tk, primarily the text widget, the treeview widget, and the canvas widget.

20.1 Scroll bars

Tk has several scrollable widgets – those that use scroll bars. Widgets that accept a scroll bar (without too many extra steps) have the options `xscrollcommand` and `yscrollcommand`. For these, to use scroll bars in `tcltk` requires two steps: the scroll bars must be constructed and bound to some widget, and that widget must be told it has a scroll bar. This way changes to the widget can update the scroll bar and vice versa. Suppose `parent` is a container and `widget` has these options, then the following will set up both horizontal and vertical scroll bars.

```
xscr <- ttkscrollbar(parent, orient = "horizontal",
                    command = function(...) tkxview(widget, ...))
yscr <- ttkscrollbar(parent, orient = "vertical",
                    command = function(...) tkyview(widget, ...))
```

The `tkxview` and `tkyview` functions set the part of the widget being shown.

To link the widget back to the scroll bar, 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 scroll bars can appear anywhere, the conventional place is on the right and lower side of the parent. The following adds scroll bars using the grid manager. The combination of weights and stickiness below will cause the scroll bars to 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)
```

This is a bit tedious, but it does give the programmer some flexibility in arranging scroll bars. For subsequent usage, we turn the above into the function `addScrollbars` (not shown). In base Tk, there are no simple means to hide scroll bars when they are not needed, although the `tcltk2` package has some code that may be employed for that.

20.2 Multiline text widgets

The `tktext` widget creates a multiline text-editing widget. If constructed with no options but a parent container, the widget can have text entered into it by the user:

```
window <- tkoplevel()
tkwm.title(window, "Simple tktext example")
txt_widget <- tktext(window)
addScrollbars(window, txt_widget)
```

The text widget is not a themed widget; hence it 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 arguments `width` and `height` are there to set the initial size, with values specifying number of characters and number of lines (not pixels; to convert see Section 17.3). 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 whether 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 whether the undo mechanism should be used. If so, the control-z keyboard shortcut or the subcommand `tktext edit` can be used to undo a change

Inserting text Inserting text can be done through the `tktext insert` subcommand. This shows how we can use `\n` to add new lines:

```
tkinsert(txt_widget,
         "1.0",
         paste("Lorem ipsum dolor",
              "sit amet,", sep = "\n"))
```

Images and other windows can be added to a text buffer, but we do not discuss that here. The value "1.0" is an index (described below) marking the beginning of the buffer.

Getting text The *tktext* `get` subcommand is used to retrieve the text in the buffer. We specify what part of the text buffer should be returned using indices. The following shows how to retrieve the entire contents of the buffer:

```
value <- tkget(txt_widget, "1.0", "end")
as.character(value) # wrong way
```

```
[1] "Lorem" "ipsum" "dolor" "sit" "amet,"
```

```
tclvalue(value)
```

```
[1] "Lorem ipsum dolor\nsit amet,\n"
```

The return value is of class `tclObj`. The above example shows that coercion to character should be done with `tclvalue` and not `as.character` to preserve the distinction between spaces and line breaks.

Indices As with the entry widget, several commands take indices to specify position within the text buffer. Only, for the multiline widget are both a line and character needed in some instances. These indices can be specified in many ways. We 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 first row and first character). In general, we 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. 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.

The specification

```
value <- tkget(txt_widget, "1.0", "end")
```

uses the index `1.0` to refer to the beginning of the buffer and the mark `"end"` to refer to the character after the end.

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 further 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 one line under the point. The values `linestart`, `lineend`, `wordstart`, and `wordend` are also available. So, `insert linestart` refers the beginning of the line from the insert point, `end -1 wordstart` refers to the beginning of the last

word in the buffer and `end - 1` chars `wordend` refers to the ending of the last word in the buffer. (The end index refers to the character just after the new line, so we go back two steps.)

Deleting text The text between two indices can be deleted using `tkdelete`, as with `tkdelete(txt_widget, "1.0", "end")`, which would clear the entire buffer's contents.

Panning the buffer: `tksee` After text is inserted, the visible part of the buffer may not be what is desired. The `tktext see` subcommand 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 can 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 by appending to the argument list, as with:

```
tkinsert(txt_widget, "end", "last words", "lastWords")
```

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 with the `first_word` tag:

```
tktag.add(txt_widget, "first_word",  
          "1.0 wordstart", "1.0 wordend")
```

The `tktext tag configure` can be used to configure properties of the tagged characters. For example:

```
tktag.configure(txt_widget, "first_word", foreground = "red",  
               font = "helvetica 12 bold")
```

There are several other configuration options for a tag. From within an R session, 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 (assuming 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 whether there is a current selection, the following can be used:

```
has_selection <- function(W) {  
  ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))  
  length(ranges) > 1 || ranges != ""  
}
```

Cut, copy, and paste The cut, copy, and paste commands are implemented through the Tk 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, we have:

```
tcl("tk_textCut", txt_widget)
```

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 pop-up menus, as in this code example:

```
popup_context <- function(W, x, y) {
  ## or use sprintf("@%s,%s", x, y) for "current"
  cur <- tkget(W, "current wordstart", "current wordend")
  cur <- tclvalue(cur)
  popup_context_menu_for(cur, x, y)      # some function
}
```

To assign a new mark, we use the `tktext` mark set subcommand specifying a name and a position through an index. Marks refer to spaces between characters. The gravity of the mark can be left or right. When it is 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(txt_widget, "leftlimit", "insert")
tkmark.gravity(txt_widget, "leftlimit", "left") # keep on left
tkinsert(txt_widget, "insert", "new text")
tkget(txt_widget, "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 right-most one changes as more text is inserted, so it would be a poor choice here.

The edit command The subcommand `tktext` edit can be used to undo text. As well, it can be used to test whether the buffer has been modified, as follows:

```
tcl(txt_widget, "edit", "undo")          # no output
tcl(txt_widget, "edit", "modified")     # 1 = TRUE
```

```
<Tcl> 1
```

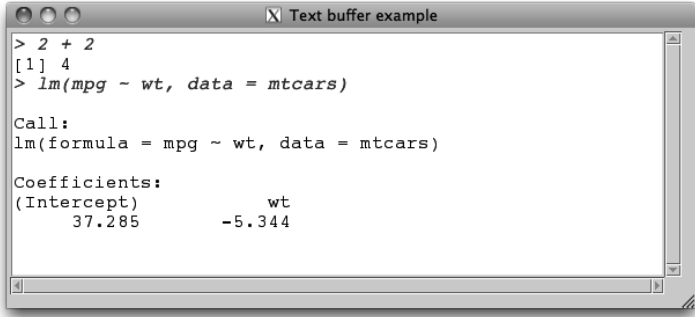


Figure 20.1: A text widget used to show formatted R commands and their output.

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 percent substitution `k` gives the keycode and `K` the key symbol as a string (`N` is the decimal number).

Example 20.1: 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. 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 (Figure 20.1).

```
w <- tktoplevel(); tkwm.title(w, "Text buffer example")
f <- ttkframe(w, padding = c(3,3,12,12))
tkpack(f, expand = TRUE, fill = "both")
txt <- tktext(f, width = 80, height = 24) # default size
addScrollbars(f, txt)
```

To distinguish between commands and their output we define the following tags:

```
tktag.configure(txt, "commandTag", foreground = "blue",
                font = "courier 12 italic")
tktag.configure(txt, "outputTag", font = "courier 12")
tktag.configure(txt, "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.

```
eval_cmd_chunk <- function(txt, cmds) {
```

```

cmd_chunks <- try(parse(text = cmds), silent = TRUE)
if(inherits(cmd_chunks, "try-error")) {
  tkinsert(t, "end", "Error", "errorTag") # add markup tag
}

for(cmd in cmd_chunks) {
  cutoff <- 0.75 * getOption("width")
  dcmd <- deparse(cmd, width.cutoff = cutoff)
  command <-
    paste(getOption("prompt"),
          paste(dcmd, collapse = paste("\n",
                                       getOption("continue")), sep = "")),
          sep = "", collapse = "")
  tkinsert(txt, "end", command, "commandTag")
  tkinsert(txt, "end", "\n")
  ## output, should check for errors in eval!
  output <- capture.output(eval(cmd, envir = .GlobalEnv))
  output <- paste(output, collapse = "\n")
  tkinsert(txt, "end", output, "outputTag")
  tkinsert(txt, "end", "\n")
}
}

```

This is how it can be used.

```
eval_cmd_chunk(txt, "2 + 2; lm(mpg ~ wt, data = mtcars)")
```

20.3 Menus

Menu bars and pop-up menus in Tk are constructed with `tkmenu`. The parent argument depends on what the menu is to do. A top-level menu bar, such as appears at the top of a window has a top-level window as its parent; a submenu of a menu bar uses the parent menu; and a pop-up menu uses a widget.

The menu widget in Tk has an option to be “torn off.” This feature 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 behavior. Otherwise, the following command will do so globally:

```
tcl("option", "add", "*tearOff", 0) # disable tearoff menus
```

A top-level menu bar is attached to a top-level window using `tkconfigure` to set the menu option 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 top-level window. Testing for native Mac OS X may be done via the following function:

```
using_Mac <- function()
  as.character(tcl("tk", "windowingsystem")) == "aqua"
```

The `tkpopup` function facilitates the creation of a pop-up menu. This function has arguments for the menu bar, and for the position where the menu should be popped up. For example, the following code will bind a pop-up 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 callback
if (using_Mac()) {
  tkbind(button, "<Button-2>", doPopup)      # right click
  tkbind(button, "<Control-1>", doPopup)    # Control + click
} else {
  tkbind(button, "<Button-3>", doPopup)
}
```

Adding submenus and action items Menus show a hierarchical 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, using the `tkmenu` `add cascade` subcommand. The option label is used to label the menu and the menu option to specify the submenu.

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:

Commands An action item is one associated with a command. The simplest proxy is a button in the menu that activates a command when selected with the mouse. The `tkmenu` `add command` allows us to specify a label, a command and optionally an image with a value for compound to adjust its layout. Action commands can possibly be called for different widgets, so the use of percent substitution is problematic. We can also specify that a keyboard shortcut be displayed through the option `accelerator`, but a separate callback must listen for this combination.

Checkboxes Action items may also be proxied by 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 callback when the checked state is toggled. The initial state is set by the value in the Tcl variable.

Radio buttons Additionally, action items may be presented through 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 against the menu's current labels.

Set state The `state` option is used to retrieve and set the current state of a menu item. This value is typically `normal` or `disabled`, the latter to indicate that the item is not available. The state can be set when the item is added or configured after the 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 20.2: Simple menu example

This example shows how we might make a very simple code editor using a text-entry widget. We use the `svMisc` package, as it defines a few GUI helpers that we use.

```
library(svMisc) # for some helpers
showCmd <- function(cmd) {
  writeLines(captureAll(parseText(cmd)))
}
```

We begin with a simple GUI comprised of a top-level window containing the text-entry widget.

```
window <- tkoplevel()
tkwm.title(window, "Simple code editor")
frame <- ttkframe(window, padding = c(3,3,12,12))
tkpack(frame, expand = TRUE, fill = "both")
text_buffer <- tktext(frame, undo = TRUE)
addScrollbars(frame, text_buffer)
```

Using `tkmenu`, we create a top-level menu bar, `menu_bar`, and attach it to our top-level window. Following that, we make “file” and “edit” submenus.

```
menu_bar <- tkmenu(window)
tkconfigure(window, menu = menu_bar)
#
file_menu <- tkmenu(menu_bar)
tkadd(menu_bar, "cascade", label = "File", menu = file_menu)
#
edit_menu <- tkmenu(menu_bar)
tkadd(menu_bar, "cascade", label = "Edit", menu = edit_menu)
```

To these sub-menu bars, we add action items. First we create a command to evaluate the contents of the buffer:

```
tkadd(file_menu, "command", label = "Evaluate buffer",
      command = function() {
        cur_val <- tclvalue(tkget(text_buffer, "1.0", "end"))
        showCmd(cur_val)
      })
```

then a command to evaluate just the current selection:

```
tkadd(file_menu, "command", label = "Evaluate selection",
      state = "disabled",
      command = function() {
        cur_sel <- tclvalue(tkget(text_buffer,
                                 "sel.first", "sel.last"))
        showCmd(cur_sel)
      })
```

and finally, we end the file menu with a separator and quit action:

```
tkadd(file_menu, "separator")
tkadd(file_menu, "command", label = "Quit",
      command = function() tkdestroy(window))
```

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")
tkimage.create("photo", "::img::undo", file = img)
tkadd(edit_menu, "command", label = "Undo",
      image = "::img::undo", compound = "left",
      state = "disabled",
      command = function() tcl(text_buffer, "edit", "undo"))
tkadd(edit_menu, "command", label = "Redo", state = "disabled",
      command = function() tcl(text_buffer, "edit", "redo"))
```

For updating the GUI, we want to configure the menu items to reflect whether the current buffer has a selection or can undo or redo. To check the selection we have:

```

tkbind(text_buffer, "<<Selection>>", function(W) {
  hasSelection <- function(W) {
    ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
    length(ranges) > 1 || ranges != ""
  }
  ## configure using an index
  sel_state <- ifelse(hasSelection(W), "normal", "disabled")
  tkentryconfigure(file_menu, 2, state = sel_state)
})

```

To check for do and undo, we bind to the Modified virtual event.

```

tkbind(text_buffer, "<<Modified>>", function(W) {
  ## not really can_undo/can_redo but nothing suitable
  can_undo <- as.logical(tcl(W,"edit", "modified"))
  undo_state <- ifelse(can_undo, "normal", "disabled")
  sapply(c("Undo", "Redo"), function(i) # match pattern
    tkentryconfigure(edit_menu, i, state = undo_state))
})

```

We add a shortcut entry to the menu bar and a binding to the top-level window for the keyboard shortcut for “undo.”

```

if(using_Mac()) {
  tkentryconfigure(edit_menu, "Undo", accelerator="Cmd-z")
  tkbind(window, "<Option-z>", function() {
    tcl(text_buffer, "edit", "undo")
  })
} else {
  tkentryconfigure(edit_menu, "Undo", accelerator="Control-u")
  tkbind(window, "<Control-u>", function() {
    tcl(text_buffer, "edit", "undo")
  })
}

```

To illustrate pop-up menus, we define one within our text widget that will grab all functions that complete the current word, using the completion function from the `svMisc` package to provide the completions. The use of `current wordstart` and `current wordend`, below, to find the word at the insertion point isn't quite right for R, as it stops at periods, but we don't pursue fixing this.

```

do_popup <- function(W, X, Y) {
  cur <- tclvalue(tkget(W, "current wordstart",
    "current wordend"))
  tcl(W, "tag", "add", "popup", "current wordstart",
    "current wordend")
  possible_vals <- head(completion(cur)[,1, drop=TRUE], n=20)
  if(length(possible_vals) > 1) {
    popup <- tkmenu(text_buffer) # create menu for popup
  }
}

```

```
sapply(possible_vals, function(i) {
  tkadd(popup, "command", label=i, command = function() {
    tcl(W,"replace", "popup.first", "popup.last", i)
  })
})
tkpopup(popup, X, Y)
}}
```

For a pop-up, 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 (!using_Mac()) {
  tkbind(text_buffer, "<Button-3>", do_popup)
} else {
  tkbind(text_buffer, "<Button-2>", function(W,X,Y) {
    ## UNIX legacy re mouse-2 click for selection copy
    tcl("clipboard","clear",displayof = W)
    do_popup(W,X,Y)
  }) # right click
  tkbind(text_buffer, "<Control-1>", do_popup) # Ctrl+click
}
```

20.4 Treeview widget

The themed treeview widget can be used to display rectangular data, like a data frame, or hierarchical data, like a list. The usage is similar, but for a minor change to indicate the hierarchical structure.

Rectangular data

The `ttktreeview` constructor creates the tree widget. There is no separate model for this widget, as there is in GTK+ or Qt, but there is a means to adjust 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 `display-columns` is used to control which of the columns are actually displayed. The default is `"all"`, but a vector of indices or names can be given.

The size of the widget is specified two ways. The `height` argument is used to adjust the number of visible rows. The requested width of the widget is determined by the combined widths of each column, whose adjustments are mentioned later.

If `frame` is a frame, then the following call will create a treeview widget with just one column showing 25 rows at a time, like the older, non-themed, list-box widget of Tk.

```

treeview <-
  ttktreeview(frame,
             columns = 1,          # column identifier is "1"
             show = "headings",  # not "#0"
             height = 25)
addScrollbars(frame, treeview)  # our scroll bar function

```

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 it has the arguments `xscrollcommand` and `yscrollcommand` for specifying scroll bars.

Adding values Rectangular data has a row and column structure. In R, data frames are stored internally by column vectors, so each column may have its own type. The treeview widget is different: it stores all data as character data and we interact with the data row by row.

Values can be added to the widget through the `ttktreeview insert parent item [text] [values]` subcommand. This requires the specification of a parent (always the root "" for rectangular data) and an index for specifying the location of the new child among 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.

In the example, this is how we can add a list of possible CRAN mirrors to the treeview display.

```

x <- getCRANmirrors()
Host <- x$Host
shade <- c("none", "gray")          # tag names
for(i in seq_along(Host))
  ID <- tkinsert(treeview, "", "end",
                values = as.tclObj(Host[i]),
                tag = shade[i %% 2]) # none or gray
tktag.configure(treeview, "gray", background = "gray95")

```

For filling in each row's content the `values` option is used. If there is a single column, like the current example, care needs to be taken when adding a value. The call to `as.tclObj` prevents the widget from dropping values after the first space.¹ Otherwise, we can pass a character vector of the proper length.

There are a number of other options for each row. If column #0 is present, the `text` option is used to specify the text for the tree row, and

¹As does wrapping the values within braces.

the option `image` can be given to specify an image to place to the left of the text value. Finally, we mention the `tag` option for `insert` that can be used to specify a tag for the inserted row. This allowed the use of the subcommand `ttktreeview` tag `configure` to configure the background color. In addition, we can adjust foreground color, font, or image for an item.

Column properties The 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(treeview, "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, we 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, columns 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 `treeview`:

```
tcl(treeview, "column", 1, width = 400,
    stretch = TRUE, anchor = "w")
```

Example 20.3: A convenience function for populating a table

We put the above commands together into a convenience function for subsequent use. The following assumes `m` is a character matrix. It returns a list containing the enclosing frame and the `treeview` object.

```
populate_rectangular_treeview <- function(parent, m) {
  enc_frame <- ttkframe(parent)
  frame <- ttkframe(enc_frame)
  tkpack(frame, expand = TRUE, fill = "both")
  treeview <- ttktreeview(frame,
    columns = seq_len(ncol(m)),
    show = "headings")
  addScrollbars(frame, treeview)
  tkpack.propagate(enc_frame, FALSE) # size from frame
}
```

```

## headings, widths
font_measure <- tcl("font", "measure", "TkTextFont", "0")
charWidth <- as.integer(tclvalue(font_measure))
sapply(seq_len(ncol(m)), function(i) {
  tcl(treeview, "heading", i, text = colnames(m)[i])
  tcl(treeview, "column", i,
      width = 10 + charWidth*max(apply(m, 2, nchar)))
})
tcl(treeview, "column", ncol(m), stretch = TRUE)
## values
if(ncol(m) == 1) m <- as.matrix(paste("{", m, "}", sep=""))
apply(m, 1, function(vals)
  tcl(treeview, "insert", "", "end", values = vals))
##
return(list(treeview = treeview, frame = enc_frame))
}

```

The use of `tkpack.propagate` allows us to control the size of the enclosing component by configuring the size of the enclosing frame. Otherwise, in the computation for requested size, the treeview widget will respond with a width computed by its column widths. However, we use a horizontal scroll bar to avoid this.

To use this we need to configure the size of the scrollable frame widget. For example:

```

window <- tkoplevel()
m <- sapply(mtcars, as.character)
a <- populate_rectangular_treeview(window, m)
tkconfigure(a$treeview, selectmode = "extended") # multiple
tkconfigure(a$frame, width = 300, height = 200) # frame size
tkpack(a$frame, expand = TRUE, fill = "both")

```

Item IDs Each row has a unique item ID generated by the widget when a row is added. The base ID is "" (why this was previously specified for the value of parent for rectangular data). For rectangular displays, the list of all IDs can be found through the `tkttreeview` children subcommand, 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 `tkttreeview` index command returns the row index.

```

children <- tcl(treeview, "children", "")
(children <- head(as.character(children))) # as.character

```

```
[1] "I001" "I002" "I003" "I004" "I005" "I006"
```

```
sapply(children, function(i) tclvalue(tkindex(treeview, i)))
```



```
I001 I002 I003 I004 I005 I006
"0"  "1"  "2"  "3"  "4"  "5"
```

Retrieving values The *ttktreeview* `item` subcommand can be used to get the values and other properties stored for each row. We specify the item and the corresponding option:

```
x <- tcl(treeview, "item", children[1], "-values") # no tkitem
as.character(x)
```

```
[1] "Universidad Nacional de La Plata"
```

The value returned from the `item` command can be difficult to parse, as Tcl places braces around values with blank spaces. The coercion through `as.character` works much better at extracting the individual columns. A possible alternative to using the `item` command, is to 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 needed. 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.

Selection The user may select one or more rows with the mouse, as controlled by the option `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 *ttktreeview* `select` command will return the current selection. The current selection marks zero, one or more than one items if "extended" is given for the `selectmode` argument. If converted to a string using `as.character` this will be a character vector of the selected item IDs. Further subcommands `set`, `add`, `remove`, and `toggle` can be used to adjust the selection programatically.

For example, to select the first six children, we have:

```
tkselect(treeview, "set", children)
```

To toggle the selection, we have:

```
tkselect(treeview, "toggle", tcl(treeview, "children", ""))
```

Finally, the selected IDs are returned with:

```
IDs <- as.character(tkselect(treeview))
```

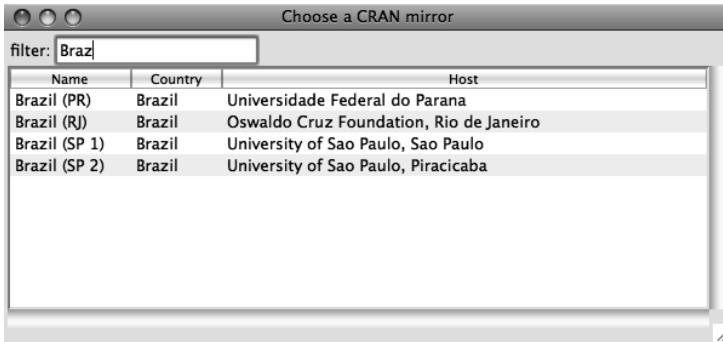


Figure 20.2: Using `ttktreeview` to show various CRAN sites. This illustration adds a search-like box to filter what repositories are displayed for selection.

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.

Within a key or mouse event callback, the clicked-on column and row can be identified by position, as illustrated in this example callback.

```
callback_example <- function(W, x, y) {
  col <- as.character(tkidentify(W, "column", x, y))
  row <- as.character(tkidentify(W, "row", x, y))
  ## now do something ...
}
```

Example 20.4: Filtering a table

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.

```
frame_0 <- ttkframe(frame); tkpack(frame_0, fill = "x")
label <- ttklabel(frame_0, text = "filter:")
tkpack(label, side = "left")
filter_var <- tclVar("")
filter_entry <- ttkentry(frame_0, textvariable = filter_var)
tkpack(filter_entry, side = "left")
```

The treeview will only show the first three columns of the data frame, although we store the fourth which contains the URL.

```
frame_1 <- ttkframe(frame)
tkpack(frame_1, expand = TRUE, fill = "both")
treeview <- ttktreeview(frame_1, columns = 1:ncol(DF),
                      displaycolumns = 1:(ncol(DF) - 1),
                      show = "headings",      # not "tree"
                      selectmode = "browse") # single selection
addScrollbars(frame_1, treeview)
```

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(treeview, "heading", i, text = nms[i])
  tcl(treeview, "column", i, width = widths[i],
      stretch = TRUE, anchor = "w")
}
```

The treeview widget does not do filtering internally.² As such we will replace all the values when filtering. This following helper function is used to fill in the widget with values from a data frame.

```
fillTable <- function(treeview, DF) {
  children <- as.character(tcl(treeview, "children", ""))
  for(i in children)
    tcl(treeview, "delete", i)          # out with old
  shade <- c("none", "gray")
  for(i in seq_len(nrow(DF)))
    tcl(treeview, "insert", "", "end", tag = shade[i %% 2],
        text = "",
        values = unlist(DF[i,]))      # in with new
  tcltag.configure(treeview, "gray", background = "gray95")
}
```

The initial call populates the table from the entire data frame.

```
fillTable(treeview, 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.

```
cur_ind <- 1:nrow(DF)
tkbind(filter_entry, "<KeyRelease>", function(W, K) {
  val <- tclvalue(tkget(W))
  poss_vals <- apply(DF, 1, function(...)
    paste(..., collapse = " "))
  ind <- grep(val, poss_vals)
```

²The model-view-controller architecture of GTK+ and Qt, makes this task much easier, as it allows for an intermediate proxy model.

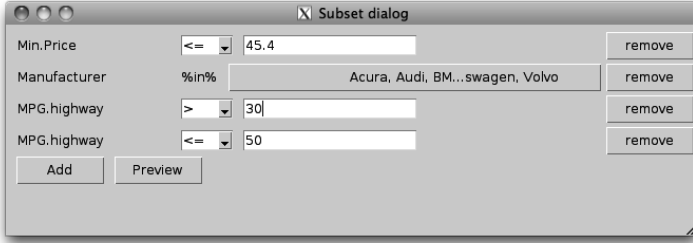


Figure 20.3: A dialog for subsetting a data frame. The example introduces a reference class to contain an unknown number of items, all of which are instances of an item class.

```
if(length(ind) == 0) ind <- 1:nrow(DF)
fillTable(treeview, 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(treeview, "<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))
})
```

Example 20.5: A dialog for subsetting a data frame

This longish example creates a framework for showing a list of similar items whose length is uncertain. There are several uses of such a framework. For example, a GUI for formulas might have items given by terms between + values, or a GUI for `ggplot2` might have items that represent individual layers of a plot. Here we use the framework to create a dialog for the subset argument of the `subset` function.³ That argument combines an arbitrary number of statements that produce logical values to produce a logical index for a data frame. For our framework, each item will produce one of these logical statements, and our list will hold the items.

To implement this, we first create a `FilterList` class. Our class has a few properties: `DF` to hold the data frame; `l` to hold the list items; `id` to

³The authors' would like to thank Liviu Andronic for ideas related to this example.

hold an internal counter to reference the list items by; and `frame` to hold a `ttkframe` instance, the parent container for each item.

```
setOldClass("tkwin")
setOldClass("tclVar")
FilterList <- setRefClass("FilterList",
                        fields = list(
                            DF = "data.frame",
                            l = "list",
                            id = "ANY",
                            frame = "tkwin"
                        ))
```

The main interface for a filter list is limited. For management, we define a method to add a list item and one to remove a list item. We also need a method (`get_value`) for analyzing the items and producing a logical vector with which to subset the data frame. Beyond that we have methods to set up the GUI, a preview method to see the current subsetting, and a method to select a variable from the data frame.

First, we define a method to set up our GUI. Here, into a parent container that is passed in, we pack in a frame (`enc_frame`) to hold the pieces of our GUI.⁴ These consist of a frame to hold the items and a frame to hold the buttons. We use the `tkgrid` layout manager, which allows us to grow the top frame as needed, yet have the buttons receive the additional expanding space.

```
FilterList$methods(
  setup_gui = function(parent) {
    enc_frame <- ttkframe(parent, padding = 5)
    tkpack(enc_frame, expand = TRUE, fill = "both")
    frame <-<- ttkframe(enc_frame)
    button_frame <- ttkframe(enc_frame)
    ## use grid to manage these
    tkgrid(frame, sticky = "news")
    tkgrid(button_frame, sticky = "new")
    tkgrid.rowconfigure(enc_frame, 1, weight = 1)
    tkgrid.columnconfigure(enc_frame, 0, weight = 1)
    ##
    add_button <-
      ttkbutton(button_frame, text = "Add",
                command = function() .self$add())
    preview_button <-
      ttkbutton(button_frame, text = "Preview",
                command = function() .self$preview())
```

⁴This means that `tkpack` needs to be used to manage any other children of this parent. An alternative would be to pass back the enclosing frame object so that it can be managed as the user desires.

```

##
  sapply(list(add_button, preview_button), tkpack,
         side = "left", padx = 5)
})

```

The initialize method simply initializes our fields and then sets up the GUI. As the point of this is to filter a data frame, the DF argument has no default value and must be specified.

```

FilterList$methods(
  initialize = function(DF, parent, ...) {
    initFields(DF = DF, l = list(), id = 0L)
    setup_gui(parent)
    callSuper(...)
  })

```

Before showing a filter, we force the user to select a variable by which to filter. This selection involves choosing one from possibly many. A table is an excellent choice for this, as it gracefully handles many values. This convenience method provides a table selection widget in a modal dialog window. Selection happens when a user selects one of the rows of the table.

```

FilterList$methods(
  select_variable = function() {
    "Return a variable name from the data frame"
    x <- sapply(DF, function(i) class(i)[1])
    m <- cbind(Variables = names(x), Type = x)
    window <- tktoplevel()
    fr <- ttkframe(window, padding = c(3,3,3,12))
    tkpack(fr, expand = TRUE, fill = "both")
    ##
    a <- populate_rectangular_treeview(fr, m)
    tkconfigure(a$frame, width = 300, height = 200)
    tkpack(a$frame, expand = TRUE, fill = "both")
    ## select a value, store in out
    out <- NA
    tkbind(a$str, "<<TreeviewSelect>>", function(W) {
      sel <- tcl(W, "selection")
      val <- tcl(W, "item", sel, "-values")
      assign("out", as.character(val)[1],
            inherits = TRUE)
      tkdestroy(window)
    })
    tkwait.window(window)
    return(out)
  })

```

Our main add method has a few tasks: to select a variable, to create a new filter item, to create a container, to do the internal bookkeeping, and

finally to call the items `make_gui` method. The `newFilterItem` call is an S3 generic used as a factory method to find the correct filter item reference class to produce an appropriate filter for the variable.

```
FilterList$methods(  
  add = function(variable_name, ...) {  
    if(missing(variable_name))  
      variable_name <- select_variable()  
    x <- get(variable_name, DF)  
    ## new item  
    id <<- id + 1  
    item <- newFilterItem(x, variable_name, id, .self)  
    ## make frame  
    enc_frame <- ttkframe(frame)  
    tkpack(enc_frame,  
           expand = TRUE, fill = "both", pady = 2)  
    l[[as.character(id)]] <<- list(frame = enc_frame,  
                                  item = item)  
  
    item$make_gui(enc_frame)  
  })
```

To remove an object requires us to remove it from our internal list and from the GUI. We use `tkpack` to manage the items, so `tkpack.forget` is used to remove the item. In the `add` method we store the enclosing frame to make this task easy.

```
FilterList$methods(  
  remove=function(id_obj, ...) {  
    "Remove. id is character or item object"  
    if(!is.character(id_obj))  
      id_obj <- id_obj$id  
    tkpack.forget(l[[id_obj]]$frame)  
    l[[id_obj]] <<- NULL  
  })
```

Here we query all the items and combine them to create a logical index vector. The item interface described below will provide its own `get_value` method, so this task is a matter of combining the results of each of those calls. We use `all` here, but if we wanted to extend this GUI, one area would be to allow the user to specify “and” or “or” between each item.

```
FilterList$methods(  
  get_value = function() {  
    "Return logical value for all filter items"  
    if(length(l) == 0)  
      return(rep(TRUE, length=nrow(DF)))  
    ##  
    out <- sapply(l, function(i) i$item$get_value())  
    out[is.na(out)] <- FALSE ## coerce NA to FALSE
```

```

    apply(out, 1, all)
  })

```

The `get_value` method makes it easy to provide a preview method to show the current state of the subsetting. Basically, we just need to create a character matrix that we want to display and then use our previously defined `populate_rectangular_treeview` function.

```

FilterList$methods(
  preview = function() {
    "Preview data frame"
    ind <- get_value()
    if(!any(ind)) {
      message("No matches")
      return()
    }
    ## coerce to character
    m <- DF[ind,]
    for(i in seq_along(m))
      m[,i] <- as.character(m[,i])
    ##
    window <- tktoplevel()
    fr <- ttkframe(window, padding = c(3,3,3,12))
    tkpack(fr, expand = TRUE, fill = "both")
    a <- populate_rectangular_treeview(fr, m)
    tkconfigure(a$frame, width = 400, height = 300)
    tkpack(a$frame, expand = TRUE, fill = "both")
    ##
    button <- ttkbutton(fr, text = "dismiss",
                        command=function() tkdestroy(window))
    tkpack(button, anchor = "sw")
    tkwait.window(window)
  })

```

To use this new class, we would integrate it into a dialog. The basic call needed would be something along the lines of the following:

```

window <- tktoplevel()
require(MASS)
filter_list <- FilterList$new(DF = Cars93, parent = window)

```

But before that will work, we need to define the filter item classes.

Filter items As mentioned, we use an S3 generic to select the reference class to provide the appropriate filter item. These are still to be defined, but we show the default choice.

```

newFilterItem <- function(x, nm = deparse(substitute(x)), id,
                          list_ref) UseMethod("newFilterItem")

```



```
newFilterItem.default <- function(x,nm=deparse(substitute(x)),
                                id, list_ref) {
  FilterItemNumeric$new(x = x, nm = nm, id = id,
                        list_ref = list_ref)
}
```

A filter item needs to produce a logical vector used for indexing. At a minimum, we require a few properties: `x` to store the variable's data that we are considering; `nm` to store the name of this variable; `id` to store the id of where this item is stored in the filter list; and `list_ref` to store a reference to the filter list.

```
FilterItem <- setRefClass("FilterItem",
                          fields = list(
                            x = "ANY",
                            nm = "character",
                            id = "character",
                            list_ref = "ANY"
                          ))
```

The filter item interface is not complicated. The most important method is `get_value` to return a logical variable. This was called by the filter list's similarly named `get_value` method. As well, we call the item's `make_gui` method in the filter list. The last method is simply a `remove` method that calls back up into the `remove` method of the item's parent filter list.

```
FilterItem$methods(
  initialize = function(...) {
    initFields(...)
    .self
  },
  get_value = function() {
    "Return logical value of length x"
    stop("Must be subclassed")
  },
  remove = function() list_ref$remove(.self),
  make_gui = function(parent, ...) {
    "Set up GUI, including defining widgets"
    remove_button <- ttkbutton(parent, text="remove",
                               command = function() {
                                 .self$remove()
                               })
    tkpack(remove_button, side = "right")
  })
```

The interesting things happen in the subclasses. For numeric values we add two new properties to help with our `get_value` method: one to store an inequality operator and one to store an expression the user can enter.

```
FilterItemNumeric <- setRefClass("FilterItemNumeric",
```

```

        contains = "FilterItem",
        fields = list(
            ineq_variable = "tclVar",
            value_variable = "tclVar"
        ))

```

With these two properties, our `get_value` method becomes a matter of pasting together an expression then evaluating it. We evaluate this within the data frame so that `value_variable` could use variable names from the data framed.

```

FilterItemNumeric$methods(
  get_value = function() {
    xpr <- paste(nm, tclvalue(ineq_variable),
                tclvalue(value_variable))
    eval(parse(text = xpr),
          envir = list_ref$DF, parent.frame())
  })

```

Our GUI has three widgets: a label, a combo box for the inequality, and an entry widget to put in values. We could simplify this, say with a slider to slide through the possible values, but using an entry widget gives more flexibility in the specification. We see that we simply pack these widgets into the parent that is passed in to the method call.

```

FilterItemNumeric$methods(
  make_gui = function(parent) {
    ## standard width for label
    label_width <- max(sapply(names(list_ref$DF), nchar))
    label <- ttklabel(parent, text=nm, width=label_width)
    ## ineq combo
    vals <- c(">=", ">", "==", "!=", "<", "<=")
    ineq_variable <<- tclVar("<=")
    ineq <- ttkcombobox(parent, values = vals,
                        textvariable = ineq_variable, width = 4)
    ## entry
    value_variable <<- tclVar(max(x, na.rm = TRUE))
    val <- ttkentry(parent, textvariable = value_variable)
    ##
    sapply(list(label, ineq, val), tkpack, side = "left",
           padx = 5)
    callSuper(parent)
  })

```

The character selection class, also used with factors, is more involved. Our `get_value` method is basically `x %in% cur_vals`, where `cur_vals` is a selection from all possible values. We might want to use a group of checkboxes here, but that can get unwieldy when there are more than a

handful of choices.⁵ We opt instead for a table-selection widget. That can take up vertical screen space. To avoid this we use a button that shows the currently selected values, and that can be clicked to open a dialog to adjust these values. To keep a consistent horizontal size to these buttons we “ellipsize” the button’s text in the `ellipsize` method. Some graphical toolkits, but not Tk, have built-in “ellipsize” methods that prove useful when controlling space allocations when translations are involved, as these can vary widely in the number of characters needed to display.

For our new subclass, we have four additional properties, the tree view for selection, the button, and vectors to store the possible values and the currently selected values.

```
FilterItemCharacter <-
  setRefClass("FilterItemCharacter",
    contains = "FilterItem",
    fields = list(
      tr = "tkwin",
      button = "tkwin",
      poss_vals = "character",
      cur_vals = "character"
    ))
```

As mentioned, our `get_value` method is easy to define:

```
FilterItemCharacter$methods(
  get_value = function() {
    x %in% cur_vals
  })
```

The main work is in our `select_values_dialog`, defined below. We use the following helper function to preselect the currently selected values when the dialog is opened.

```
sel_by_name <- function(tr, nms) {
  all_ind <- as.character(tcl(tr, "children", ""))
  vals <- sapply(all_ind, function(i) {
    as.character(tcl(tr, "item", i, "-values"))
  })
  ind <- names(vals[vals %in% nms])
  sapply(ind, function(i) tcl(tr, "selection", "add", i))
  sapply(setdiff(all_ind, ind),
    function(i) tcl(tr, "selection", "remove", i))
}
```

Here is our previously mentioned convenience method to make the button size uniform by “ellipsizing” the button’s label.

⁵A table of checkboxes might also be used, but this isn’t directly supported by the `treetview` widget of `tcltk`. The intrepid could set the `image` attribute for each row to show a check or non-check depending on the state.

```

FilterItemCharacter$methods(ellipsize = function() {
  tmp <- paste(cur_vals, collapse = ", ")
  if((N <- nchar(tmp)) > 50)
    tmp <- sprintf("%s...%s", substr(tmp, 0, 15),
                  substr(tmp, N-12, N))
  sprintf("%50s", tmp)
})

```

This is the main dialog for selecting values. Here multiple selection is achieved by extending the selection through holding the shift and control keys while clicking on items.

```

FilterItemCharacter$methods(
  select_values_dialog = function() {
    window <- tkoplevel()
    fr <- ttkframe(window, padding = c(3,3,12,12))
    tkpack(fr, expand = TRUE, fill = "both")
    tkpack(ttklabel(fr,
      text = "Select values by extending selection"))
    ## selection
    m <- matrix(poss_vals)
    colnames(m) <- "Values"
    a <- populate_rectangular_treeview(fr, m)
    tkconfigure(a$str, selectmode = "extended")
    tkconfigure(a$frame, width = 200, height = 300)
    tkpack(a$frame, expand = TRUE, fill = "both")

    sel_by_name(a$str, cur_vals)      # see above

    tkbind(a$str, "<<TreeviewSelect>>", function() {
      ind <- as.character(tcl(a$str, "selection"))
      cur <- sapply(ind, function(i) {
        as.character(tcl(a$str, "item", i, "-values"))
      })
      if(length(cur) == 0)
        cur <- character(0)
      cur_vals <<- cur
    })
    ## buttons
    frame_1 <- ttkframe(fr)
    tkpack(frame_1)
    toggle_button <- ttkbutton(frame_1, text="toggle",
      command=function() toggle_sel(a$str))
    set_button <- ttkbutton(frame_1, text = "set",
      command=function() tkdestroy(window))
    sapply(list(toggle_button, set_button), tkpack,
      side = "left", padx = 5)
    ## make modal

```

```
        tkwait.window(window)
        tkconfigure(button, text = ellipsize())
    })
```

Our main GUI for a character or factor item then has three widgets: labels for the name and %in% operator and a button.

```
FilterItemCharacter$methods(make_gui = function(parent) {
  poss_vals <- sort(unique(as.character(x)))
  cur_vals <- poss_vals
  ## label, ineq, val
  l_width <- max(sapply(names(list_ref$DF), nchar))
  label <- ttklabel(parent, text = nm,
                    width = l_width)
  ##
  in_label <- ttklabel(parent, text = "%in%")
  ##
  button <- ttkbutton(parent, text = ellipsize(),
                      command = .self$select_values_dialog)
  ##
  sapply(list(label, in_label), tkpack,
         side = "left", padx = 5)
  tkpack(button,
         expand = TRUE, fill = "x", side = "left")
  callSuper(parent)
})
```

We leave it as an exercise for the reader to add a subclass for logical variables or date variables.

Editable tables of data

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. An interface for this Tcl package is provided in the `tcltk2` package (`tk2edit`). The `gdf` function of `gWidgetstcltk` is based on this.

Hierarchical data

Specifying tree-like or hierarchical 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

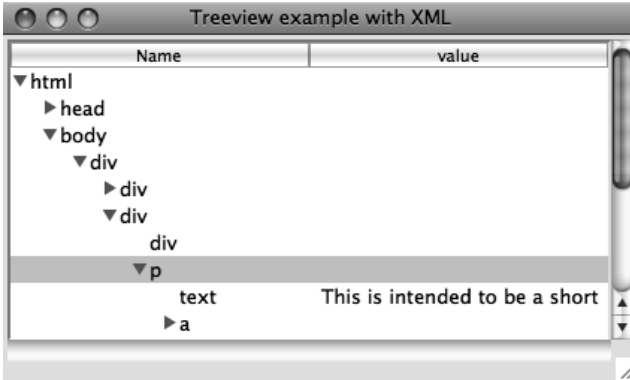


Figure 20.4: Illustration of using `ttktreeview` widget to show hierarchical data returned from parsing an HTML document with the XML package.

manner to indicate the level of ancestry they have relative to the root. To insert hierarchical data into the widget the same `ttktreeview insert` subcommand is used, except that instead of using the root item `"` as the parent item, we use 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 20.6: Using the treeview widget to show an XML file

This example shows how to display the hierarchical 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)
```

```
file_name <- "http://www.omegahat.org/RFXML/shortIntro.html"
doc <- htmlTreeParse(file_name, useInternalNodes = TRUE,
                     error = function(...) {})
root <- xmlRoot(doc)
```

Our GUI is primitive, with just a treeview instance added.

```
treeview <- ttktreeview(frame, displaycolumns = "#all",
                      columns = 1)
addScrollbars(frame, treeview)
```

We configure our column headers and set a minimum width below. Recall that the tree column is designated "#0".

```
tcl(treeview, "heading", "#0", text = "Name")
tcl(treeview, "column", "#0", minwidth = 20)
tcl(treeview, "heading", 1, text = "value")
tcl(treeview, "column", 1, minwidth = 20)
```

To map the tree-like structure of the XML document into the widget, we define the following function to add to the treeview instance recursively. We add to the value column (through the values option) only when the node does not have children. We use `do.call`, as a convenience, to avoid constructing two different calls to the insert subcommand.

```
insertChild <- function(treeview, node, parent = "") {
  l <- list(treeview, "insert", parent, "end",
           text = xmlName(node))
  children <- xmlChildren(node)
  if(length(children) == 0) { # add in values
    values <- paste(xmlValue(node), sep = " ", collapse = " ")
    l$values <- as.tclObj(values) # avoid split on spaces
  }
  tree_path <- do.call("tcl", l)

  if(length(children)) # recurse
    for(i in children) insertChild(treeview, i, tree_path)
}
insertChild(treeview, root)
```

At this point, the GUI will allow us to explore the markup structure of the XML file. We continue this example to show two things of general interest, but that are really artificial for this example.

Drag and drop First, we show how we 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 whether a drag event is ongoing.

```
.selected_id <- "" # 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(treeview, "<Button-1>", function(W, x, y) {
  .selected_id <- as.character(tcl(W, "identify", "row", x, y))
})
```

The motion callback configures the cursor to indicate a drag event and sets the dragging flag. We might also put in code to highlight any drop areas.

```
tkbind(treeview, "<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. We 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(treeview, "<ButtonRelease-1>", function(W, x, y, X, Y) {
  if(.dragging && .selected_id != "") {
    w <- tkwinfo("containing", X, Y)
    if(as.character(w) == as.character(W)) {
      dropID <- as.character(tcl(W, "identify", "row", x, y))
      try(tkmove(W, .selected_id, dropID, "0"), silent = TRUE)
    }
  }
  .dragging <- FALSE; .selected_id <- "" # 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 hierarchical structure. When there are children a recursive call is made.

```
tree_to_list <- function(treeview) {
  l <- list()
  walk_tree <- function(child, l) {
    l$name <- tclvalue(tcl(treeview, "item", child, "-text"))
    l$value <- as.character(tcl(treeview, "item", child,
                               "-values"))
  }
}
```



```
children <- as.character(tcl(treeview, "children", child))
if(length(children)) {
  l$children <- list()
  for(i in children)
    l$children[[i]] <- walk_tree(i, list()) # recurse
}
return(l)
}
walk_tree("", l)
}
```

20.5 Canvas widget

The canvas widget provides an area to display lines, shapes, images, and widgets. The canvas widget is quite complicated, and we content ourselves to describing a subset of its possibilities. For an excellent example of how it can be used to provide a useful GUI for R see the `RnavGraph` package by Waddell and Oldford.

As described on its manual page, the canvas widget implements structured graphics, displaying any number of items or objects of various types. 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-themeable widget, has many arguments, including these standard ones: `width`, `height`, `background`, `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 item. The basic shape types that we can add are "line", "arc", "polygon", "rectangle", and "oval". Their options specify the size using x and y coordinates. Other options allow us 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, we 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, 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.

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 we are 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 us 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 and scaled but not rotated. The `tkcanvas move tag_or_ID x y` subcommand implements incremental moves (where `x` and `y` specify the horizontal and vertical shift in pixels). The subcommand `tkcanvas coords tag_or_ID [coordinates]` allows us to respecify the coordinates for the item. The `tkcanvas scale` command is used to rescale items. Except for window objects, an item can be raised to be on top of the others through the `tkcanvas raise item_or_ID` subcommand.

Bindings As usual, bindings can be specified overall for the canvas through `tkbind`. However, bindings can also be set on specific items through the subcommand `tkcanvas bind tag_or_ID event function` (or with `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 20.7: Using a canvas to make a scrollable frame

This example⁶ 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 scroll bars using the `tkcanvas create window` subcommand.

There are two bindings to the `<Configure>` event. The first updates the scroll region of the canvas widget to include the entire canvas, which grows as items are added to the frame. The second binding ensures the

⁶This example is modified from an example found at <http://mail.python.org/pipermail/python-list/1999-June/005180.html>

child window is the appropriate width when the canvas widget resizes. The height is not adjusted, as this is controlled by the scrolling.

```
scrollable_frame <- function(parent, width=300, height=300) {
  canvas_widget <-
    tkcanvas(parent,
             borderwidth = 0, highlightthickness = 0,
             width = width, height = height)
  addScrollbars(parent, canvas_widget)
  #
  frame <- ttkframe(canvas_widget, padding = c(0,0,0,0))
  frame_id <- tkcreate(canvas_widget, "window", 0, 0,
                     anchor = "nw", window = frame)
  tkitemconfigure(canvas_widget, frame_id, width = width)
  ## update scroll region
  tkbind(frame, "<Configure>", function() {
    bbox <- tcl(canvas_widget, "bbox", "all")
    tcl(canvas_widget, "config", scrollregion = bbox)
  })
  ## adjust "window" width when canvas is resized.
  tkbind(canvas_widget, "<Configure>", function(W) {
    width <- as.numeric(tkwininfo("width", W))
    frame_width <- as.numeric(tkwininfo("width", frame))
    if(frame_width < width)
      tkitemconfigure(canvas_widget, frame_id, width = width)
  })
  return(frame)
}
```

To use this, we create a simple GUI as follows:

```
window <- tkoplevel()
tkwm.title(window, "Scrollable frame example")
frame <- ttkframe(window)
tkpack(frame, expand = TRUE, fill = "both")
scroll_frame <- scrollable_frame(frame, 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 17.3). 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 returned by `tkfont.families`.

```
font_families <- as.character(tkfont.families())
## skip odd named ones
font_families <- font_families[grepl("^[[[:alpha:]]",
                                     font_families)]
for(i in seq_along(font_families)) {
  font_name <- sprintf("::font::-%s", i)
```

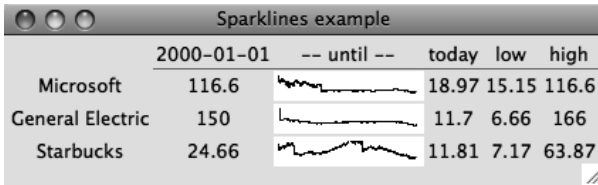


Figure 20.5: Example of embedding sparklines in a display organized using `tkgrid`. A `tkcanvas` widget is used to display the graph.

```
try(tkfont.create(font_name, family = font_families[i],
                 size = 14),
    silent = TRUE)
l <- ttklabel(scroll_frame, text = font_families[i],
              font = font_name)
tkpack(l, side = "top", anchor = "w")
}
```

Example 20.8: Using canvas objects to show sparklines

Edward Tufte, in his book *Beautiful Evidence*^[11], advocates for the use of *sparklines* – small, intense, simple datawords – to show substantial amounts of data in a limited visual space. This example shows how to use a `tkcanvas` object to display a sparkline graph using a line object.⁷ The example also uses `tkgrid` to lay out 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 such improvements as an exercise.

This function simply shortens our call to `ttklabel`. We use the global frame (a `ttkframe`) as the parent.

```
mL <- function(label) { # save some typing
  if(is.numeric(label))
    label <- sprintf("%.2f", label)
  ttklabel(frame, text = label, justify = "right")
}
```

We begin by making the table header along with a top rule.

```
tkgrid(mL(""), mL("2000-01-01"), mL("-- until --"),
       mL("today"), mL("low"), mL("high"))
tkgrid(ttkseparator(frame), row=1, column = 1, colspan = 5,
       sticky = "we")
```

This function adds a sparkline to the table. A sparkline here is just a line item, but there is some work to do, in order to scale the values

⁷The `sparkTable` package creates sparklines for documents and web pages.

[11] Edward R. Tufte. *Beautiful Evidence*. Graphics Press, 2006.

to fit the allocated space. This example uses stock market values, as we can conveniently employ the `get.hist.quote` function from the `tseries` package to get interesting data.

```
add_sparkline <- 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)
  ##
  spark_line_canvas <- tkcanvas(frame,
                               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(spark_line_canvas, "create", "line")
  sapply(seq_along(x), function(i) {
    l[[2*i + 2]] <<- x[i]
    l[[2*i + 3]] <<- y1[i]
  })
  do.call("tcl", l)

  tkgrid(mL(label), mL(start), spark_line_canvas,
         mL(end), mL(min), mL(max), pady = 2, sticky = "e")
}
```

We can then add some rows to the table as follows:

```
add_sparkline("Microsoft", "MSFT")
add_sparkline("General Electric", "GE")
add_sparkline("Starbucks", "SBUX")
```

Example 20.9: 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 = "blue")
tkadddtag(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 hovers over. 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 = "blue"))
```

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.

```
last_pos <- numeric(2) # global to track position
tag_selected <- function(W, x, y) {
  tkadddtag(W, "selected", "withtag", "current")
  tkitemraise(W, "current")
  last_pos <- as.numeric(c(x, y))
}
tkitembind(canvas, "point", "<Button-1>", tag_selected)
```

When the mouse moves, we use `tkmove` to have the currently selected point move too. As `tkmove` is parameterized by differences, we track the differences between the last position recorded and the current position.

```
move_selected <- function(W, x, y) {
  pos <- as.numeric(c(x,y))
  tkmove(W, "selected", pos[1] - last_pos[1],
         pos[2] - last_pos[2])
  last_pos <- pos
}
tkbind(canvas, "<B1-Motion>", move_selected)
```

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.

```
tkbind(canvas, "<ButtonRelease-1>",
       function(W) tkdtag(W, "selected"))
```

This page intentionally left blank

Programming Graphical User Interfaces in R

Programming Graphical User Interfaces in R introduces each of the major R packages for GUI programming: RGtk2, qtbase, Tcl/Tk, and gWidgets. With examples woven through the text as well as stand-alone demonstrations of simple yet reasonably complete applications, the book features topics especially relevant to statisticians who aim to provide a practical interface to functionality implemented in R. The book offers:

- A how-to guide for developing GUIs within R
- The fundamentals for users with limited knowledge of programming within R and other languages
- GUI design for specific functions or as learning tools

The accompanying package, ProgGUIinR, includes the complete code for all examples as well as functions for browsing the examples from the respective chapters and is available through CRAN. Accessible to seasoned, novice, and occasional R users, this book shows that for many purposes, adding a graphical interface to one's work is not terribly sophisticated or time consuming.

 **CRC Press**
Taylor & Francis Group
an informa business
www.crcpress.com
6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K12672

