Preface

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 where 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, like Windows and Mac OS, and statistical software like 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. On some platforms, such as Windows and Mac OS, there are graphical front-ends that provide a CLI through a text console control. 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 to a wide range of R's functionality and supports plugins.

The third group of users, those who only require R for certain tasks and do not wish to learn the language, are targeted by task-specific GUIs. These interfaces usually do not 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 exploRase (**?**), limmaGUI (**?**), playwith, latticist, and Rattle (**?**). Other GUIs are aimed at teaching statistics, e.g., teachingDemos. 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 nor time-consuming. This book does not attempt to cover the development of GUIs that require knowledge of another programming language, although several such projects exist. Many of these are general front-ends to R, such as the Java-based GUI JGR, the rkWard GUI for KDE, the biocep GUI written using Java and the RServe package, or even the Windows GUI that comes with R's Windows package. There are also several special purpose GUIs, like iPlots, which are largely implemented in Java, relying on rJava, a native interface between R and Java.

The bulk of this text covers four different packages for writing GUIs in R. The gWidgets package is covered first. This provides a common programming interface over several R packages that implement low-level, native interfaces to GUI toolkits. The gWidgets interface is much simpler – and less powerful – than the native toolkits, so is useful for a programmer who does not wish to invest too much time into perfecting a GUI. There are a few other packages that provide a high-level R interface to a toolkit such as rpanel or svDialogs, but we focus on 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 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+ nor Qt, these libraries come pre-installed with the Windows binary, thus avoiding installation issues for the average end-user. The bindings to Tk were the first ones to appear for R and most of the GUI projects above, notably `Rcmdr`, use this toolkit.

These four main parts are preceeded 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 `browsegWidgetsFiles`, `browseRGtk2Files`, `browseQtFiles` and `browseTclTkFiles` for browsing the examples from the respective chapters.

Contents

1

The basic ideas of Graphical User Interfaces

1.1 A simple GUI in R

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

Many GUIs can be thought of as different views of some data model. In this example, the data simply consists of information holding the state of the game, defined here in a global variable `board`.

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

A GUI contains one or more views, each of which is tied to an underlying data model. In our case, the view is the game board that we display through an R graphics device. The `layoutBoard` function creates a canvas for this view:

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

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. Here we benefit from the universal familiarity with the board game.

The user typically sends input to a GUI through 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 has a different implementation. is toolkit For our game, we will use the `locator` function built into the base R graphics system:

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

The `locator` function responds to mouse clicks. One specifies 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). Such a GUI that blocks the flow of a program contingent on 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 `doPlay`, `clickHandler` is an *event handler*. Its job is to process the output of the `locator` function, checking first if the user terminated `locator` using the keyboard. If not it proceeds to draw the move, and then, if necessary, the computer's move. Afterwards, play is repeated until there is a winner or a "cat's" game.

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

The use of `<<-` in the handler illustrates a typical issue in GUI design 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. When this is inconvenient, direct manipulation of environment objects is a viable alternative. In the above case, we simply modify the global environment, which encloses `clickHandler`.

Validation of user input is an important task for a GUI. In the above, the `clickHandler` 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 function draws either an "x" or an "o". It does so using the `lines` function. The `z` argument contains the coordinates of the square to draw.

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

One could use `text` to place a text "x" or "o", but this may not 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 if a game is finished:

```
isFinished <- 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 easily check all the possible ways to get three in a row.

This function picks a move for the computer:

```
doComputerMove <- function() {
  newMove <- sample(which(board == 0),1) # random !
  board[newMove] <<- -1
  z <- c((newMove-1) %% 3, (newMove-1) %/% 3) + 1
  drawMove(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 left over positions; we leave implementing a better strategy to the interested reader.

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

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

When the game is launched, we first lay out the board and then call `doPlay`. When `doPlay` 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 continually add features 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 adds complexity to the interface, often useful, but sometime it just increases the 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 (i.e., mouse). The WIMP approach was developed at Xerox PARC in the 1970's and later popularized by the Apple Macintosh in 1984. This is particularly evident in the separation of the window from the 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 a WIMP approach.

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

Abstractly, WIMP is a command language, where the user executes commands, often called actions, on a document by interacting with graphical controls. Every control in a window belongs to some abstract menu. Two common ways of organizing controls into menus are the 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 (**?**), but we prefer the term *dialogs*, or *dialog boxes*. These terms may also refer to smaller sub-windows that are used for alerts or confirmation. The program often needs to wait for user input before continuing with an action, in which case the window is modal. We refer to these as *modal dialog boxes*.

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

for example, many GUI frameworks employ invisible widgets to lay out the other widgets in a window.

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

In some cases, typical usage suggests one control over another. The choice of printer for each is specified through a combo box. However, for other choices a variety of widgets are employed. For example, the control to indicate the number of copies for the Mac is a simple text entry window, whereas for the KDE and R dialog it is a spinbutton. The latter minimizes user error, say through entering a non-positive integer. The KDE and Mac dialogs have icons to compactly represent actions, whereas the R example has none. The landscape icon for the Mac is very clear and provides this feature without having to use a sub dialog.

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

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

The Mac GUI provides a very nice preview of the current document indicating to the user clearly what is to be printed and how much. Adjusting GUIs to the possible state is an important user interface property. GUI areas that are not currently sensitive to user input are grayed out. For example, the "collate" feature of the GUI only makes sense when multiple copies are selected, so the designers have it grayed out until then. A common element of GUI design is to only enable controls when their associated action is possible, given the state of the application.

The Mac GUI has the number of pages in focus, whereas Windows places the printer in focus. This allows the user to interact with the GUI without the mouse. Typically the tab key is used to step through the con-

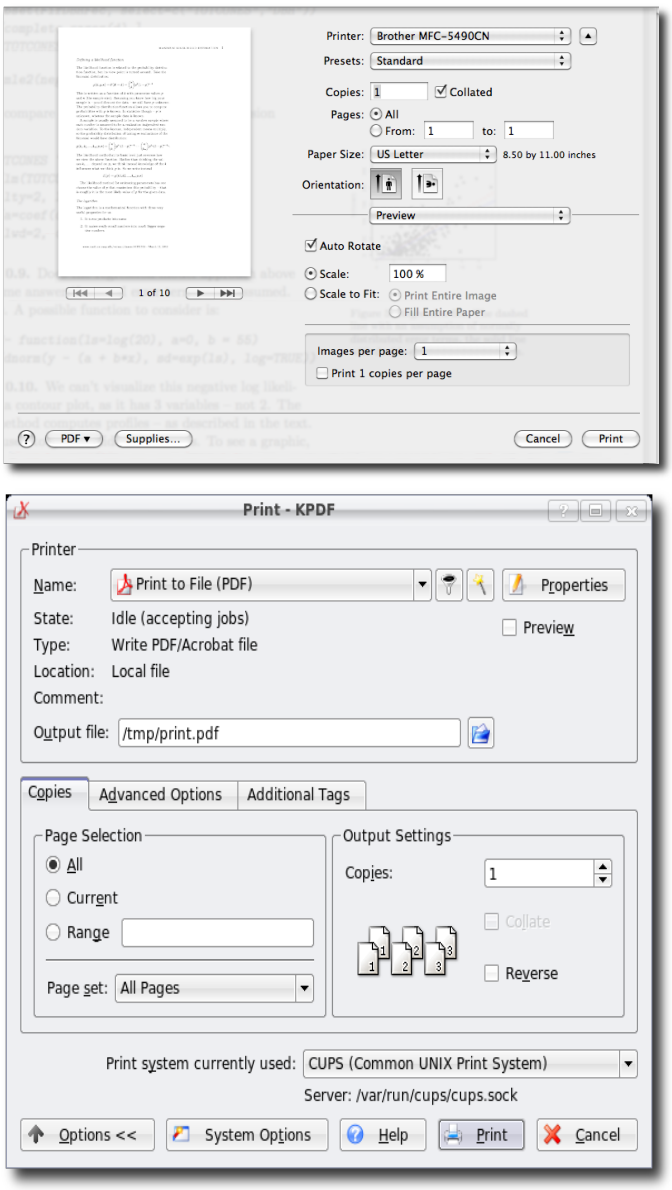

Figure 1.1: Two print dialogs. One from Mac OS X 10.6 and one from KDE 3.5.

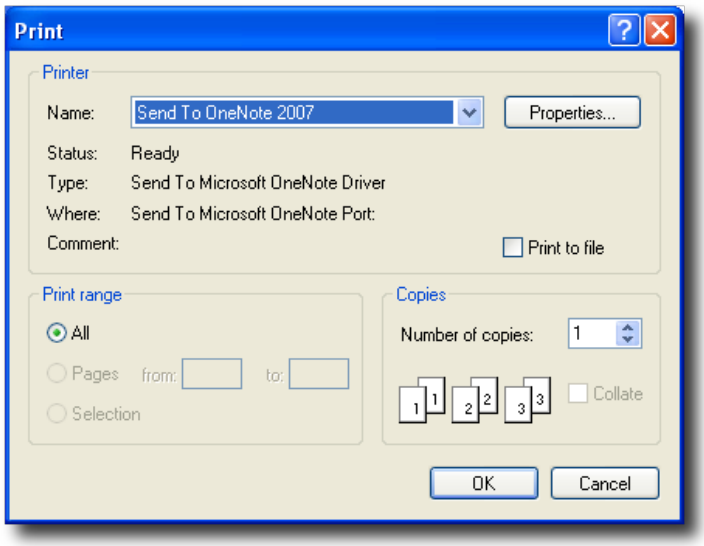

Figure 1.2: R's print dialog under windows XP using XP's native dialog.

trols. GUI's often have shortucts 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 one can't tell from this. However, one can see the use of stock icons on the buttons to guide the user.

1.3 Controls

This section provides an overview of many common controls, i.e., widgets that either 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.

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

Type of data	Single	Multiple
Boolean	Checkbox, toggle button	-
Small list	radiogroup	checkboxgroup
	combobox	list box
	list box	
Moderate list	combobox	list box
	list box	
Large list	list box, auto complete	list box
Sequential	slider spinbutton	
Tabular	table	table
Heirarchical	tree	tree

Choice of control

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

Figure 1.3 shows several such controls in a single GUI. A checkbox enables an intercept, a radio group selects either full factorial or a custom model, a 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 1.3 shows an SPSS dialog to build terms in a model. R power users may be much faster specifying the formula through a text entry box, but beginning R users coming to grips with the command line and the concept of a formula may benefit from the assistance of a well designed GUI. One might desire an interface that balances the needs of both types of user, or the SPSS interface may be appropriate. Knowing the potential user base is important.

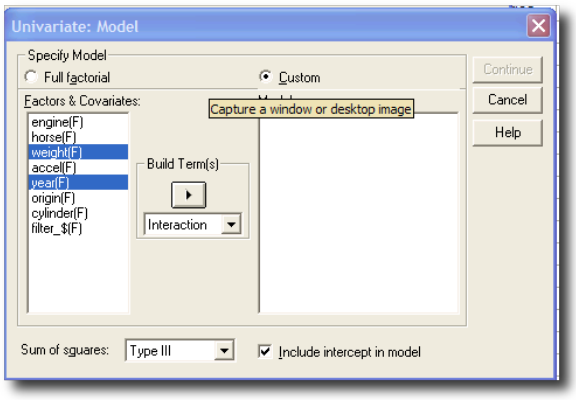

Figure 1.3: A dialog box from SPSS version 11 for specifying terms for a linear model. The graphic shows a dialog that allows the user to specify individual terms in the model using several types of widgets for selection of values, such as a radio button group, a checkbox, combo boxes, and list boxes.

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.

Radio buttond

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 to select a preset channel. When a new button was

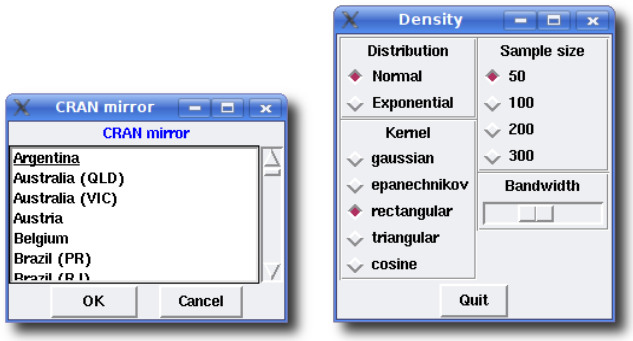

Figure 1.4: Two applications of the `tcltk` package. The left graphic is produced by `chooseCRANmirror` and uses a list box to allow selection from a long list of possibiulities. The right graphic is the `tkdensity` demo from the package. It uses radio buttons and a slider to select the parameter values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

pushed in, the previously pressed button popped up. 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 1.4 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 selects one value from several. However, a combo box only displays 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 like incremental type ahead search enhance usability.

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 them simultaneously, a scrollbar is typically provided for adjusting the visible range. Unlike the combo box, the choices are immediately visible to the user. Figure 1.4 shows a list box created by R that is called from the 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 spinbuttons

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 1.4) uses a slider to dynamically adjust the bandwith of a density estimate.

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. The text box can usually 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 compared to 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 1.1 to adjust the number of copies.

Inititiating 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 menubars and toolbars.

Buttons

A *button* issues commands when pressed, usually via a mouse click. In Figure 1.1, 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. (**?**) In Figure 1.3 we see how SPSS uses buttons in its dialogs: buttons which are not valid in the current state are disabled; buttons which are designed to open subsequent dialogs have trailing dots; and the standard actions of

resetting the data, canceling the dialog or requesting help are given their own buttons on the right edge of the dialog box.

To speed the user through a dialog, a button may be singled out as the default button, so its action will be called if the user presses the return key. Buttons may be given accelerator key bindings, so that their actions are executable by typing the proper key combination. The KDE print dialog in Figure 1.1 has these bindings indicated through the underlined letter on the button 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, where an icon is used to augment or replace a text label on a button, a toolbar, in a list box, etc. When icons appear on toolbars and buttons, they are associated with actions, so an icon should be a pictorial representation of an action. The choice of icon has a significant impact on usability.

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 viewed, for example, as the active data frame, a report, or a graphic.

The styles used for menu bars are fairly standardized, as this allows new users to quickly orient themselves with a GUI. The visible menu names are often in the order `File`, `Edit`, `View`, `Tools`, then application specific menus, and finally a `Help` menu. Each visible menu item when clicked opens a menu of possible actions. The text for these actions conventionally use 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 key-board accelerator, such as `Find Next F3` indicating that both "N" as a keyboard accelerator and F3 as a shortcut will initiate this same action. (Shortcuts are not translated, but keyboard accelerators must be. As such, their use is less so. 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, they be disabled, or greyed 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 also allow the user to set properties or attributes of current state of the GUI. There may be checkboxes drawn next to the menu item or some icon indicating the current state.

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

Toolbars

Toolbars are used to give immediate access to the frequently used actions defined in the 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, some save dialog is summoned, or some 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 keyboard accelerator, and a handler to call when the action is selected. When a particular action is not possible due to the state of the GUI, it should be disabled, so that the associated widgets are not sensitive to user interaction.

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, for example asking for confirmation when overwriting a file. Modal dialog boxes can be disruptive to the flow of interaction, so 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 called during an interactive R session, the user is unable to interact with the command line until a file has been specified and the dialog dismissed.

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 to dismiss the dialog, often labeled "OK." Additional buttons/responses are possible. The *confirmation dialog* variant would add a "Cancel" button which invalidates 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.

Displaying data

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

Tabular display

A *table widget* shows tabular data, such as a data frame, where each column has a specific data type and cell rendering strategy. Table widgets handle the display, sorting and selection of records from a dataset. Depending on the configuration of the widget, cells may be editable. Figure 1.5 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, like that in a data frame. Other widgets support the display of more complex data structures. If the data has a heirarchical structure, then a *tree widget* may be appropriate for

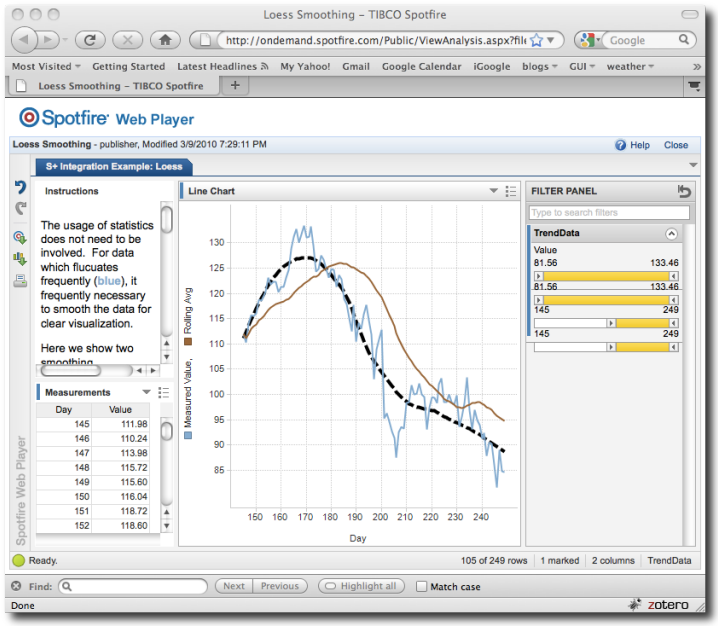

Figure 1.5: A screen shot from Tibco's Spotfire web player illustrating a table widget (lower left), displaying the cases that are summarized in the graphic. The right bar filters the cases in the table.

its display. Examples of heirarchical data in R are directory structures, the components of a list, or class hierarchies. The object browser in `JGR` uses a tree widget to show the components of the objects in a users session (the left graphic of Figure 1.6). The root node of the tree is the "data" folder, and each data object in the global workspace is treated as an offspring of this root node. For the data frame `iraq`, its variables are considered as offspring of the data frame. In this case these variables have no further offspring, as indicated by the "page" icon.

Displaying and editing text

The letter `P` in `WIMP` stands for "pointer," so it is unsurprising 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 both type and move the mouse at the same time. For statistical GUIs, especially when integrating with the command-line interface of R, the flexibity 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.

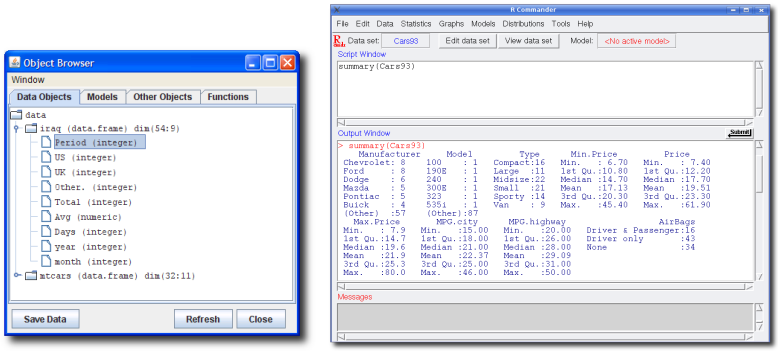

Figure 1.6: Some windows from R GUIs. The left graphic shows the object browser in the JGR GUI using a tree widget to display the possibly heirarchical nature of R objects. The right graphic shows the main Rcmdr (1.3-11) window illustrating the use of multi-line text entry areas for a command area, an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

Single line text

A text entry widget for editing a single line of text is found in the KDE print dialog (Figure 1.1). 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 edit boxes

The right graphic of Figure 1.6 shows three multi-line 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 and the user through the GUI and to display feedback and status messages.

Labels

A label is a widget for placing text into a GUI that is typically not intended for editing, or even selecting with a mouse. The main role of a label is to describe another component of the GUI. Most toolkits support rich text in labels. Figure 1.6 shows labels marked in red and blue in `tcltk`.

Statusbars

A statusbar displays general status messages, as well as feedback on actions initiated by the user, such as progress or errors. In the traditional document-oriented GUI, statusbars are placed at the bottom.

Related to status bar are info bars or alert boxes, that allow a programmer to leave a transient message for the user usually just below the toolbar.

Tooltips

A tooltip is a small window that is displayed when a user hovers their mouse over a tooltip-enabled widget. These 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 to the user that something is happening.

1.4 Containers

The KDE pring dialog of Figure 1.1 contains many of the widgets we discussed in the previous section. Before we can create such a dialog, we need to introduce how to position widgets on the screen. This process is called *widget layout*.

A layout emerges from the organization of the widgets into a hierarchy, where 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 simply assign 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 to the window size dictated by the window manager. Thus, it is strongly encouraged to delegate the responsibility of layout to a *layout manager* that 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.

Containers

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 menu bar, a tool bar and a status bar. The primary content of the window is inserted between the tool bar 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 represent 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. Placing 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.

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 1.1 contains two frames, visually grouping widgets by their function: either `Page Selection` or `Output Settings`.

Tabbed notebooks

A notebook represents each of its children as 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 once. The tabbed notebook is a space efficient, categorizing container that is most appropriate when a user is only interested in one page at a time. Modern web browsers take advantage of it 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 multitude of options into simple categories: “Copies”, “Advanced Options”, and “Additional Tags”.

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 occassionally useful and are only of interest to a small subset of the users. For example, the `Options` button in Figure 1.1 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, according to a present need. 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 two panes, either vertically (stacked) or horizontally. The area separating the two panes, sometimes called a *sash*, can be adjusted by dragging it with the mouse.

Layout algorithms

Box layout

The box container is perhaps the simplest and most common type of layout container. A box will pack its children either horizontally or vertically. 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 1.7 illustrates this.

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 requirement.

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 1.7 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 1.7).

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 1.7. One could think of this space 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 fully show its label with some fixed space placed between the buttons. If the dialog is expanded, it is seen that there is a spring between the 3rd and 4th buttons, so that the first 3 are aligned with the left side of the window and the last two the right side.

Grid layout

The box layout algorithm only aligns its children along a single dimension. The horizontal box, for example, vertically aligns its children. Nevertheless, nesting permits the construction of complex layouts using only simple boxes. However, it is sometimes desirable to align widgets in both dimensions, i.e., to lay them out on a grid. The most flexible grid layout

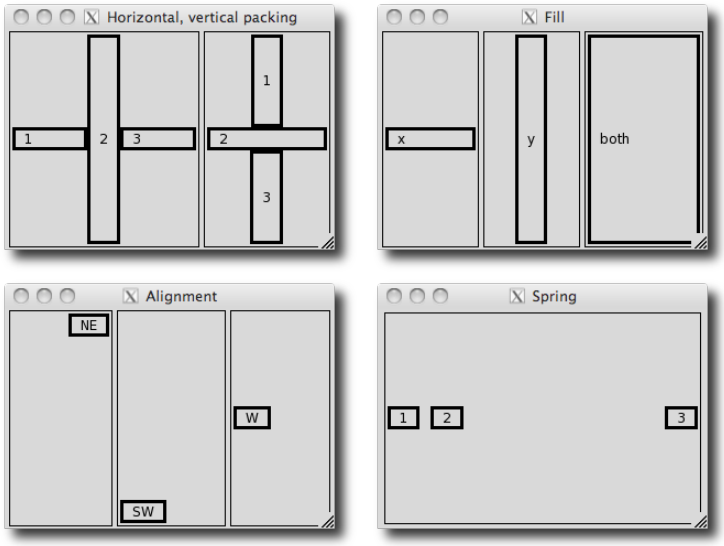

Figure 1.7: Different possibilities for packing child components within a box. The upper left shows horizontal and vertical layout. The upper right shows some possible alignments or anchorings. The lower left shows that a child could "expand" to fill the space either horizontally, vertically, or both. The lower right shows both a fixed amount of space between the children and an expanding spring between the child components.

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 their cell.

The widgets in the "Printer" frame of Figure 1.7 are subject to a grid layout with five columns and six rows. The first row begins with the "Name:" label, and each widget in that row occupy a separate column. This exposes the size of each column. The first column has only labels, with text justified to the left. The labels are aligned horizontally to each other and vertically with adjacent field.

Part I

The gWidgets package

2

gWidgets: Overview

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

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

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

2.1 Constructors

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

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

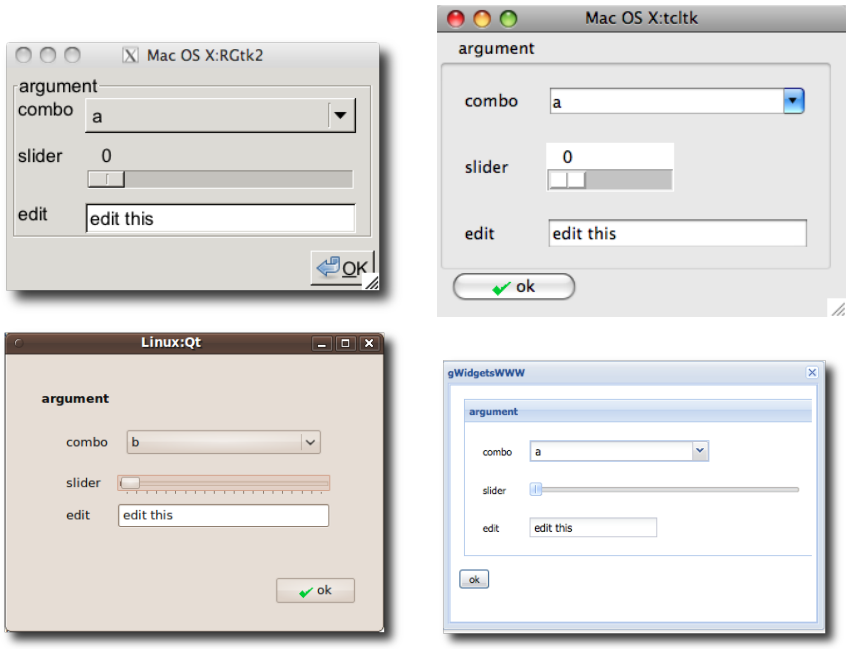

Figure 2.1: The gWidgets package works with different operating systems and different GUI toolkits. This shows, the same code using the RGtk2, tcltk, qtbase packages for a toolkit. Additionally, the gWidgetsWWW package is used in the lower right figure.

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

This example defines five different widgets, a window, a box container, a label, a single-line text edit area and a button, all produced by some constructor.

In the gWidgets API most constructors have the following form:

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

where the arguments vary depending on the object being made.

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

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

be abbreviated `cont`. This defines the GUI layout, a topic taken up in Chapter 3.

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

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

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

The constructors produce one of three general types of objects: containers, such as the top level window `w` and the box container `g` (Table 3.1); components, such as a label `l`, the edit area `e`, or the button `b`. (Tables 4 and 5); and dialogs (Table 2.4).

2.2 Methods

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

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

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

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

```
svalue(e)
```

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

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

value selected or the index of the selected value. The `index=TRUE` argument of `svalue` may be given to refer to values by their index. For these selection widgets the familiar `[` and `[<-` methods refer to the underlying data to be selected from.

The call

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

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

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

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

2.3 Callbacks

In our example, the lines

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

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

For example, a typical idiom within a callback is

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

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

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

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

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

As an example, we could have specified the button as

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

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

We passed in the object `e` through the action argument as an illustration. This is useful, as one need not worry finding `e` throug R's scoping rules in the call to `svalue`.

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

If these few methods are insufficient and toolkit-portability is not of interest, then the `addHandler` generic can be used to specify a toolkit-specific signal and a callback.

2.4 Dialogs

The `gWidgets` package provides a few constructors to quickly make some basic dialogs for showing messages or gathering information. Mostly these are modal dialogs that take control of the event loop, not allowing any other part of the GUI to be active for programmatic interaction. As such, in `gWidgets`, constructors of modal dialogs do not return an object to manipulate through its methods, but rather return the user response to the dialog. For example, the `gfile` dialog, described later, is a modal dialog that pops up a means to select a file returning the selected file path or `NA`. It is used along the lines of:

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

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

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

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

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

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

Method	Description
`addHandlerChanged`	Primary handler call for when a widget's value is "changed." The interpretation of "change" depends on the widget.
`addHandlerClicked`	Sets handler for when widget is clicked with (left) mouse button. May return position of click through components `x` and `y` of the `h` list.
`addHandlerDoubleclick`	Sets handler for when widget is double clicked
`addHandlerRightclick`	Sets handler for when widget is right clicked
`addHandlerKeystroke`	Sets handler for when key is pressed. The `key` component is set to this value, if possible.
`addHandlerFocus`	Sets handler for when widget gets focus
`addHandlerBlur`	Sets handler for when widget loses focus
`addHandlerExpose`	Sets handler for when widget is first drawn
`addHandlerUnrealize`	Sets handler for when widget is undrawn on screen
`addHandlerDestroy`	Sets handler for when widget is destroyed
`addHandlerMouseMotion`	Sets handler for when widget has mouse go over it
`addDropSource`	Specify a widget as a drop source
`addDropMotion`	Sets handler to be called when drag event mouses over the widget
`addDropTarget`	Sets handler to be called on a drop event. Adds the component `dropdata`.
`addHandler`	(Not cross-toolkit) Allows one to specify an underlying signal from the graphical toolkit and handler
`removeHandler`	Remove a handler from a widget
`blockHandler`	Temporarily block a handler from being called
`unblockHandler`	Restore handler that has been blocked
`addHandlerIdle`	Call a handler during idle time
`addPopupmenu`	Bind popup menu to widget
`add3rdMousePopupmenu`	Bind popup menu to right mouse click

Table 2.3: Table of constructors for basic dialogs in gWidgets

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

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

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

For example,

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

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

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

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

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

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

This illustrates how to use the return value.

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

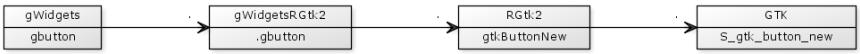

Figure 2.2: The construction of a button in `gWidgets`

Table 2.4: Installation notes for GUI toolkits.

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

2.5 Installation

The `gWidgets` package interfaces with an underlying R package through an intermediate package. For example, Figure 2.2 shows the sequence of calls to produce a button. First the `gWidgets` package dispatches to a toolkit package (`gWidgetsRGtk2`), which in turn calls functions in the underlying R package (`RGtk2`) which in turn calls into the graphical toolkit. As such, to use `gWidgets` with the GTK+ toolkit one must have installed on their computer the GTK libraries, the `RGtk2` package and the two `gWidgets` packages.

The difficulty for the end user is the installation of the graphic toolkit, as all other packages are installed through CRAN, or are recommended packages with an R installation (`tcltk`). Table 2.5 describes the installation process for different operating systems and toolkits.

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

3

gWidgets: Container Widgets

After identifying the underlying data to manipulate and how to represent it, GUI construction involves three basic steps: creation and configuration of the main components; the layout of these components; and connecting the components through callbacks to make a GUI interactive. This chapter discusses the layout process within `gWidgets`. Layout in `gWidgets` is done by placing child components within parent containers which in turn may be nested in other containers. (This is more like GTK+, and not Qt, where layout managers control where the components are displayed.) The `gWidgets` package provides a just few types of containers: top-level windows, box containers, a grid container, a paned container and a notebook container. Figure 3.1 shows most all of these employed to produce a GUI to select and then show the contents of a file.

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

3.1 Top-level windows

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

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

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

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

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

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

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

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

The initial placement of a window, $x + y$, will be decided by the window manager, unless the parent argument is specified. If this is done with a vector of x and y pixel values, the upper left corner will be placed there. The parent argument can also be another gwindow instance. In this case, the new window will be positioned over the specified window and be transient for the window. That is, it will be disposed when the parent window is. This is useful, say, when a main window opens a dialog window to gather values.

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

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

Handlers Windows objects can be closed programmatically through their dispose method. Windows may also be closed through the window manager, by clicking a close icon in the title bar. The handler argument is called just before the window is destroyed, but cannot prevent that from

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

Table 3.1: Constructors for container objects

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

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

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

In most GUIs, the use of menubars, toolbars and statusbars is often reserved for the main window, while dialogs are not decorated so. In `gWidgets` it is suggested, although not strictly enforced unless done so by the underlying toolkit, that these be added only to a top-level window. We discuss these widgets later in Section 4.5.

A modal window

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

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

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

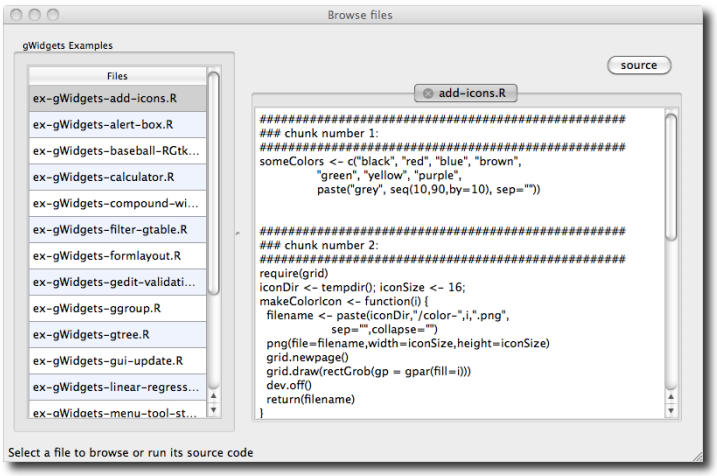

Figure 3.1: The example browser for gWidgets showing different layout components.

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

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

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:

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

Table 3.2: Container methods

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

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

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

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

These arguments are (Figure 3.2 shows combinations of these arguments under `gWidgetsQt`):

expand, fill When `expand=TRUE` is specified then the space allocated for the child widget will expand in some manner.

fill, anchor When a widget is placed into is allocated space, it can be desirable that it grow to fill the available space. The `fill` argument, taking a value of `x`, `y` or `both` indicates how the widget should expand. For many toolkits, the default fill is orthogonal to the direction of packing.

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

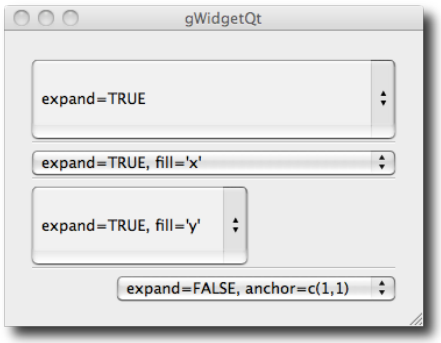

Figure 3.2: Different combinations of `expand`, `fill` and `anchor` for combo boxes in `gWidgetsQt`. The `fill` and `anchor` arguments may be overridden by the underlying toolkit for some widgets.

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

Spacing For spacing between the child components, the 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 .

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

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

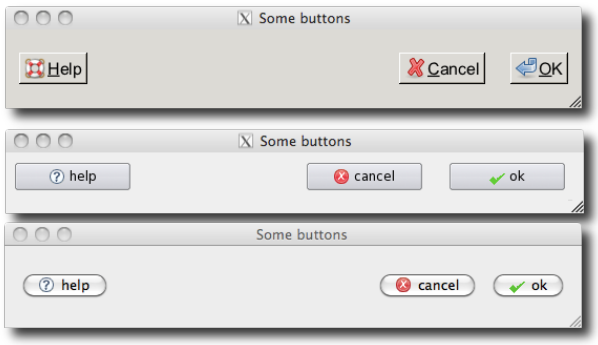

Figure 3.3: Button layout for `RGtk2` (top), `tcltk` (middle) and `qtbase` (bottom). Although the same code is used for each, the different styling yields varying sizes.

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

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

The `gframe` and `gexpandgroup` containers

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

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

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

The basic framed container is used along these lines:

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

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

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

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

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

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

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

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 layout a simple form for collecting information as follows:

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

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

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

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

The [method may be used to return the child occupying position i,j. To return the main properties of the widgets just defined can be done through:

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

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

3.4 Paned containers: the gpanedgroup container

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

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

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

A simplified version of the layout in Figure 3.1 would be

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

```
files <- list.files(d)
#
w <- gwindow("gpanedgroup example", visible=FALSE)
pg <- gpanedgroup(cont = w)
tbl <- gtable(files, cont=pg)           # left side
t <- gtext("", cont=pg, expand=TRUE)    # right side
visible(w) <- TRUE
svalue(pg) <- 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 `gWidgets` constructor has a few arguments, not all supported by each toolkit. The argument `tab.pos` is used to specify the location of the tabs using a value of 1 through 4 with 1 being botton, 2 left side, 3 top and 4 right side being used, with the default being 3 (similar numbering as used in `par`). The `closebuttons` argument takes a logical indicating whether the tabs should have close buttons on them. In this case, the argument `dontCloseThese` can be used to specify which tabs, by index, should not be closable.

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

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

From some viewpoint, the notebook widget is viewed as a vector with a names attribute (the labels) and components being the child components. As such, the `[` method returns the 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.

```
w <- gwindow("gnotebook example")
```

```
nb <- gnotebook(cont=w)
```

New pages are added as follows:

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

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

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

To remove the current page

```
dispose(nb)
```

4

gWidgets: Control Widgets

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

Buttons

The button widget allows a user to initiate an action through clicking on it. Buttons have labels, conventionally verbs indicating action, and often icons. The `gbutton` constructor has an argument `text` to specify the text. For text that matches the stock icons of `gWidgets` (Section 4) an icon will appear. (The `ok` button below, but not the custom `par...` one.)

In common with the other controls, the argument `handler` is used to specify a callback and the `action` argument will be passed along to this callback (unless it is a `gaction` object, whose case is described in Section 4.5). The default handler is the click handler which can be specified at construction, or afterward through `addHandlerClicked`. The underlying toolkit's method of invoking a callback through keyboard navigation is used.

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

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

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

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

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

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

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

The button's label is its main property and can be queried or set with svalue or svalue<-. Most GUIs will make a button insensitive to user input if the button's action is not currently permissible. Toolkits draw such

buttons in a grayed-out state. As with other components, the `enabled<-` method can set or disable whether a widget can accept input.

Labels

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

The `font<-` method can also be used to set the text markup (Table 4.1). For some of the underlying toolkits, setting the argument `markup` to `TRUE` allows a native markup language to be used (GTK+ had PANGO, Qt has rich text).

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

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

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

HTML text

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

Statusbars

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

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

Displaying icons and images stored in files

The `gWidgets` package provides a few stock icons that can be added to various GUI components. A list of the defined stock icons is returned by the function `getStockIcons`. The names attribute defines the valid stock icon names. It was mentioned that if a button's label 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. [1] The file to display is specified through the `filename` argument of the constructor. This value is combined with that of the `dirname` argument to specify the file path. Stock icons are specified by using their name for the `filename` argument and the character string `"stock"` for the `dirname` argument. [2]

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

The default handler is a button click handler.

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

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

More stock icon names may be added through the function `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 to select a color from, as an alternative to a more complicated color chooser dialog. [3]

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

[2] For `gWidgetsRGtk2`, the size of a stock icon can be adjusted through the `size` argument, with a value from `"menu"`, `"small_toolbar"`, `"large_toolbar"`, `"button"`, or `"dialog"`.

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

We begin by defining 16 arbitrary colors.

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

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

```
require(grid)
iconDir <- tempdir(); iconSize <- 16;
makeColorIcon <- function(i) {
  filename <- sprintf("%s%scolor-%s.png", iconDir,
                      .Platform$file.sep, i)
  png(file=filename, width=iconSize, height=iconSize)
  grid.newpage()
  grid.draw(rectGrob(gp=gpar(fill=i)))
  dev.off()
  return(filename)
}
```

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

```
icons <- sapply(someColors, makeColorIcon)
iconNames <- sprintf("color-%s", someColors)
addStockIcons(iconNames, icons)
```

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

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

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

4.1 Text editing controls

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

Single-line, editable text

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

A simple usage might be:

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

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

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

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

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

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

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

Example 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 where typically a GUI expects just characters or numbers. As such, it is helpful to indicate to the user if their value is a valid expression. This example shows how to implement a validation framework on a single-line edit widget so that the user has feedback when an expression will not evaluate properly. When the value is invalid we set the text color to red.

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

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

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

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

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

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

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

Multi-line, editable text

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

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

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

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

a GUI that resembles a calculator. Such a GUI may offer familiarity to new R users, although certainly it is no replacement for a command line.

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

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

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

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

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

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

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

4.2 Selection controls

A common task for a GUI control is to select a value or values from a set of numbers or a table of numbers. Figure 4.1 shows a simple GUI for the `EBImage` package allowing a user to adjust a few of the image properties using various selection widget.

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

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

Checkbox widget

The simplest selection control is the `checkbox` widget that allows the user to set a state as `TRUE` or `FALSE`. The constructor has an argument `text` to set a label and `checked` to indicate if the widget should initially be checked. The default is `TRUE` (there is no third, uncommitted state as possible in some toolkits). By default the label will be drawn aside a box with a check, if the argument `use.togglebutton` is `TRUE`, a toggle button – which appears depressed when `TRUE` – is used instead.

In Figure 4.1 a toggle button is used for "Thresh" and could be constructed as

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

The `svalue` method returns a logical indicating if the widget is in the checked state. Use `svalue<-` to set the state. The label's value is returned

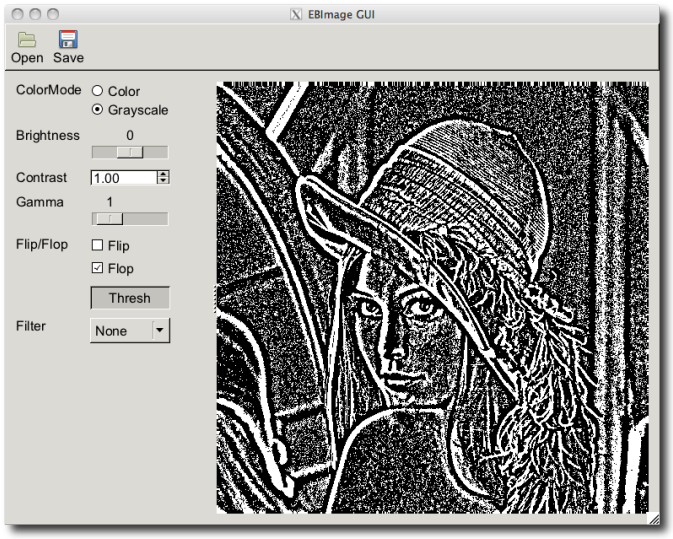

Figure 4.1: A simple GUI for the `EBImage` package illustrating many selection widgets

by the `[` method, and can be adjusted through `[<-`. (We take the abstract view that the user is selecting, or not, from the length-1 vector, so `[` is used to set the data to select from.)

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

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

Radio button widget

A radio button group allows the user to choose one of a few items. A radio button group object is returned by `gradio`. The items to choose from are specified as a vector of values to the `items` argument (2 or more). These items may be displayed horizontally or vertically (the default) as specified by the `horizontal` argument which expects a logical. The `selected` argument specifies the initially selected item, by index, with a default of the first.

In Figure 4.1 a radio button is used for "ColorMode" and could be constructed as

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

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

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

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

In Figure 4.1 a group of check boxes is used to allow the user to "flip" or "flop" the image. It could be created with

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

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

That is, each of these has the same effect:

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

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

A combo box

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

The combo box in Figure 4.1 could be done through:

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

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

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

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

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

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

A slider control

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

In Figure 4.1 a slider is used to update the brightness. The call is similar to:

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

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

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

A spin button control

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

In Figure 4.1 a spin button is used to adjust the contrast, a numeric value. The following will reproduce it

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

Selecting from the file system

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

The "Open" button in Figure 4.1 is bound to this action:

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

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

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

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

Selecting a date

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

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

4.3 Display of tabular data

The `gtable` constructor produces a widget that displays data in a tabular form from which the user can select one (or more) rows. The widgets performance under `gWidgetsRGtk2` and `gWidgetsQt` is much faster and able to handle larger data stores than under `gWidgetstcltk`, as there is no native table widget in Tcl/Tk. All perform well on moderate-sized data sets (10 or so columns and fewer than 500 rows). [4]

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

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

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

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

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

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

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

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

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

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

to the column headers, and `dim` and `length` the underlying dimensions of the data store.

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

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

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

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

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

})
```

Filtering The arguments `filter.column` and `filter.FUN` allow one to specify whether the user can filter, or limit, the display of the values in the data store. The simplest case is if a column number is specified to the `filter.column` argument. In which case a combo box is added to the widget with values taken from the unique values in the specified column. Changing the value of the combo box restricts the display of the data to just those rows where the value in the filter column matches the combo box value. More advanced filtering can be specified using the `filter.FUN` argument. If this is a function, then it takes arguments `(data_frame, filter.by)` where the data frame is the data, and the `filter.by` value is the state of a combo box whose values are specified through the argument `filter.labels`. This function should return a logical vector with length matching the number of rows in the data frame. Only rows corresponding to `TRUE` values will be displayed. If `filter.FUN` is the character string "`manual`" then the `visible<-` method can be used to control the filtering, again by specifying a logical vector of the proper length. See Example 4.5 for an application.

The `gtable` widget shows clearly the trade offs between using `gWidgets` and a native toolkit under R. As will be seen in later chapters, setting up a table to display a data frame using the toolkit packages directly can involve a fair amount of coding as compared to `gtable`, which makes it very easy.

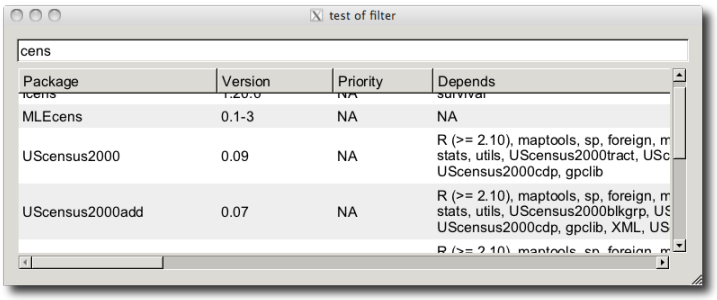

Figure 4.2: Example of using a filter to narrow the display of tabular data

However, gWidgets provides far less functionality. For example, there is no means to adjust the formatting of the displayed text, or to embed other widgets into the tabular display, such as check boxes.

Example 4.4: Simple filtering
We use the Cars93 data set from the MASS package to show how to set up a display of the data which provides simple filtering based on the type of car, whose value is stored in column 3.

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

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

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

Example 4.5: More complex filtering

Even with moderate-sized data sets, the number of rows can be quite large, in which case it is inconvenient to use a table for selection unless some means of searching or filtering the data is used. This example uses the many possible CRAN packages, to show how a gedit instance can be used as a search box to filter the display of data (Figure 4.2). The addHandlerKeystroke method is used so that the search results are updated as the user types.

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

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

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

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

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

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

```
ourMatch <- function(curVal, vals) {
  grepl(curVal, vals)
}
```

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

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

Example 4.6: Using the "observer pattern" to write a workspace view
This example takes the long way to make a workspace browser. (The short way is `gvarbrowser`.) The goal is to produce a GUI that will allow the user to view the objects in their current workspace. We would like these views to by dynamic though – when the workspace changes we would like the views to update. Furthermore, we may want to have different views, such as one for functions and one for data sets. Such requirements lead us to the observer design pattern in programming, where each view is notified of changes to an "observable."

The observer design pattern uses observables, objects which record a list of observers and notify them when there is a change, and observers. We will use an observable for a data model to store information about

our current workspace and the views will be the observers. The basic observable pattern can be coded using reference classes as:

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

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

This can get more involved (we implement signals or we could allow observers to be blocked, etc.), but we keep it simple for this example.

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

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

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

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

```
          if(notify)
            notify_observers(model=.self)
          invisible()
        }))
```

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

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

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

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

```
require(digest)
update_fun <- function() {
  "update vector of ws_objects if applicable"
  x <- sort(ls(envir=.GlobalEnv))
  ## filter out envRefClass objects --
  isRef <- function(i)
    is(get(i, envir=.GlobalEnv), "envRefClass")
  x <- x[!sapply(x, function(i) isRef(i))]
  ds <- sapply(x, function(i)
               digest(get(i, envir=.GlobalEnv)))

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

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

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

Now we create our reference class.

```
setRefClass("WSModel",
            contains="Model",
            fields=list(
              ws_objects="character",
              ws_objects_digest="character"
              ),
            methods=list(
              update=update_fun,
              get_objects=get_objects))
```

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

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

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

```
                "Update view as model has updated"
              }))
```

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

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

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

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

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

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

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

```
makeDataFrame <- function(x, envir=.GlobalEnv) {
  d <- data.frame(variable=character(0),
                  size=character(0), description=character(0),
```

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

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

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

To illustrate the flexibility of this framework, we also define a subclass to show just the data frames in a combo box. The combo box is a bit contrived in this example, but having a means to select a data frame is common to R GUIs.

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

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

```
w <- gwindow()
```

```
nb <- gnotebook(cont=w)
#
m <- getRefClass("WSModel")$new()
#
view <- TableView$new(parent=nb, model=m, label="data")
view$klass <- c("factor", "numeric", "character",
                "data.frame", "matrix", "list")
#
view1 <- TableView$new(parent=nb, model=m,
                       label="not a function")
view1$klass <- "!function"
#
view2 <- TableView$new(parent=nb, model=m, label="all")
view3 <- DfView$new(parent=nb, model=m, label = "data frames")
#
m$update()                                  # notifies views
svalue(nb) <- 1
```

4.4 Display of hierarchical data

The `gtree` constructor can be used to display hierarchical structures, such as a file system or the components of a list. The `gtree` widget is populated dynamically by computing the child components. A parameterizion of the data to be displayed is in terms of the path of the node that is currently selected. The `offspring` argument is assigned a function of two arguments, the path of a particular node and the arbitrary object passed through the optional `offspring.data` argument. This function should return a data frame with each row referring to an offspring for the node and whose first column is a key that identifies each of the offspring.

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

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

```
offspring <- function(path=character(0), lst, ...) {
  if(length(path))
    obj <- lst[[path]]
  else
      obj <- lst
```

```
  nms <- names(obj)
  hasOffspring <- sapply(nms, function(i) {
    newobj <- obj[[i]]
    is.recursive(newobj) && !is.null(names(newobj))
  })
  data.frame(comps=nms, hasOffspring=hasOffspring,
             stringsAsFactors=FALSE)
}
```

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

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

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

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

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

For some toolkits, the column type must be determined prior to rendering. 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 4.7: Using `gtree` to explore a recursive partition

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

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

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

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

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

```
offspring <- function(key, offspring.data) {
  if(missing(key) || length(key) == 0)  # which 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 4.3)

```
w <- gwindow("Example of gtree")
```

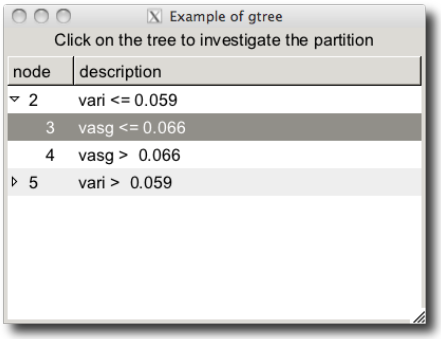

Figure 4.3: GUI to explore return value of a model fit by the `party` package.

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

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

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

4.5 Actions, menus and toolbars

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`, `parent` and the standard `handler` and `action`. The label appears as the text on a button, the menu item or toolbar text, whereas the icon will decorate the same if possible. For some toolkits, the tooltip pops up when the mouse hovers. (See also the `tooltip<-` method.) The key accelerator will bind a keyboard shortcut, such as Control-s to an action. The `parent` argument is used to specify a widget whose toplevel container will process the shortcut.

methods The main methods for actions are `svalue<-` to set the label text and `enabled<-` to adjust whether the widget is sensitive to user input. All proxies of the action are set through one call.

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

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

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

Toolbars

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

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

For example:

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

The `gtoolbar` constructor takes the list as its first argument. As toolbars belong to the window, the corresponding `gWidgets` objects use a `gwindow` object as the parent container. (Some of the toolkits relax this to allow other containers.) The argument `style` can be one of `"both"`, `"icons"`, `"text"`, or `"both-horiz"` to specify how the toolbar is rendered.

Menubars, popup menus

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

A example of such a list might be

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

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

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

Popup menus Popup menus can be created for a right click event through the `add3rdMousePopupmenu` constructor. (Or control-button-1 for Mac OS X.) This constructor has arguments `obj` to specify a widget, like a button, to initiate the popup, `menulist` to specify the menu and optionally an `action` argument.

Example 4.8: Popup menus
This example shows how to add a simple popup menu to a button.

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

5

gWidgets: R-specific widgets

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

5.1 A graphics device

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

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

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

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

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

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

This shows how the two can be used:

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

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

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 and give it a non-specific name.

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

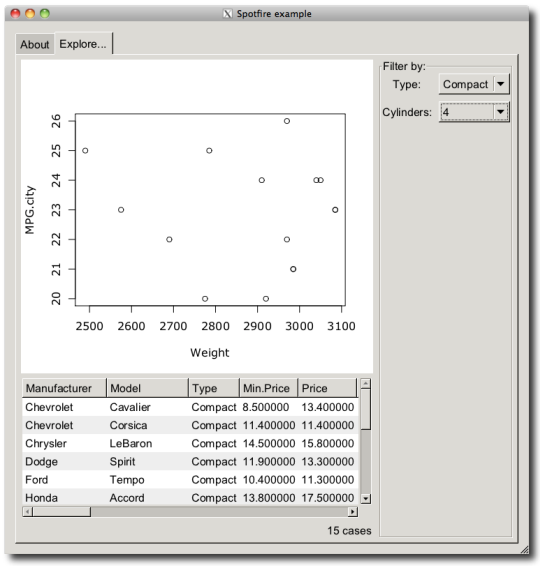

Figure 5.1: A GUI to filter a data frame and display an accompanying graphic.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
  }

  visible(tbl) <- ind                    # update table

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

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

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

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

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

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

Workspace browser

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

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

Example 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.

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

We create a function to initialize the interface.

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

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

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

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

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

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

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

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

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

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

Help browser

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

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

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

Command line widget

A simple command line widget is created by the `gcommandline` constructor. This is not meant as a replacement for any of R's commandlines, but is provided for light-weight usage. A text box allows users to type in R commands. The programmer may issue commands to be evaluated and displayed through the `svalue<-` method. The `value` assigned is a character string holding the commands. If there is a `names` attribute, the results will be assigned to a variable in the global workspace with that name. The `svalue` and `[` methods return the command history.

Simplifying creation of dialogs

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

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

Part II

The RGtk2 package

6

RGtk2: Overview

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

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

RGtk2 provides virtually all of the functionality in GTK+ to the R programmer. In addition, RGtk2 interfaces with several other libraries in the GTK+ stack: Pango for font rendering; Cairo for vector graphics; GdkPixbuf for image manipulation; 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 (camelback style).

The full API for GTK+ is quite large, and complete documentation of it is beyond our scope. However, the GTK+ documentation is algorithmically converted into the R help format during the generation of RGtk2. This conveniently allows the programmer to refer to the appropriate documentation within an R session, without having to consult a web page, such as http://library.gnome.org/devel/gtk/stable/, which lists the C API of the stable 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(x) 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 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"         "GtkObject"         "GInitiallyUnowned"
[4] "GObject"
```

For those familiar with object-oriented programming in R, the returned character vector could be interpreted as 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] "GtkBuildable"        "AtkImplementorIface"
```

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

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 GtkWindow class:

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

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

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 always must 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)
NULL
```

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. One interprets the attribute according to S3 conventions, so that the class names are in order from most to least derived:

```
class(window)
```

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

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

```
interface(window)
```

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

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

6.4 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 depends on an understanding of how of one programmatically manipulates widgets by invoking methods. Methods are functions that take an instance of their class as the first argument and instruct the widget to perform an action.

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

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

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

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

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

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

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

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

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

6.5 Properties

The GTK+ API uses properties to store object state. Properties are similar to R attributes and even more so to S4 slots. They are inherited, typed, self-describing and encapsulated, so that an object can intercept access to the underlying data. A list of properties definitions belonging to the widget 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] "related-action"         "use-action-appearance"
[3] "user-data"              "name"
[5] "parent"                 "width-request"
[7] "height-request"         "visible"
```

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

There are a few different ways to access these properties. The methods `get` and `set` get and set properties of a widget, respectively. The set funtion 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"))[[1]]
window$set(icon = image, title = "Hello World 1.0")
```

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

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

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

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

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

```
[1] FALSE
```

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

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

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

6.6 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 only lists the signals provided directly by the GtkButton. To list all inherited signals, we need to loop over the hierarchy, but it is not common to do this in practice, as the documentation includes information on the signals.

The gSignalConnect 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)
NULL
```

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

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

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

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

The after argument is a logical value indicating if 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.

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

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

Then after clicking, we would have

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

```
1 2
```

Callbacks for signals emitted by window manager events are expected to return a logical value. Failure to do so can cause errors to be raised. 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:

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

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

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, like `"toplevel"`.

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

```
GtkWindowType
```

```
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 explicitly use the enumeration vectors; specifying the nickname will work in most cases, including all method invocations, and is preferable as it is easier for human readers to comprehend.

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

```
GtkWidgetFlags
```

```
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 one to check whether a value belongs to a combination. For example, we could check whether our window is top-level:

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

```
[1] TRUE
```

6.8 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 one runs 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`:

```
g <- gtkBuildableNew()
g$addFromFile("buildable.xml")
```

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:

```
d <- g$getObject("dialog1")
d$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(w, userData) {
  print("hello world")
}
g$connectSignals()
```

The GUI should now be ready for use.

7

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 arguments `type` to specify the type of window to create. The default is a top-level window, which we will always use, as the alternative is for "popups" which are meant for internal use, e.g., for implementing menus. The second argument is `show`, which by default is `TRUE`, indicating that the window should be shown. If set to `FALSE`, the window, like other widgets, can later be shown by calling its `show` method. The `showAll` method will also show any child components. These can be reversed with `hide` and `hideAll`.

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

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

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

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

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

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

To display multiple widgets in a window, one simply needs to add a non-`GtkBin` container as the child widget.

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 the user clicks on the close button, 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(w, "delete-event", function(event) {
  gtkMessageDialog(parent=w, flags=0, type="question", buttons=c("yes", "no"),
                   "Are you sure you want to quit?")
  dlg$run() != 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:

```
w$destroy()
```

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

```
w <- gtkWindow(show=FALSE); w$setTitle("Top level window")
d <- gtkWindow(show=FALSE); d$setTitle("dialog window")
d$setTransientFor(w)
d$setPosition("center-on-parent")
d$setDestroyWithParent(TRUE)
w$show()
d$show()
```

The above code produces a non-modal dialog window from scratch. Due to its transient nature, it can hide parts of the top-level window, but, unlike a modal dialog, it does not prevent that window from receiving events. GTK+ provides a number of convenient high-level dialogs, discussed 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 explicitly, as a rectangle. However, as a user interface, a GUI is dynamic and interactive. The size constraints of widgets will change, and the window will be resized. The programmer cannot afford to explicitly manage a dynamic layout. Thus, GTK+ implements automatic layout in the form of container widgets.

Basics

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:

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

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

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

```
[1] "Hello world"
```

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

```
w[[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

```
w$remove(l)
w$add(l)
```

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.

Widget size negotiation

We have already seen perhaps the simplest automatic layout container, GtkWindow, 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:

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

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

```
l$getAllocation()$allocation
```

```
  x  y width height
 -1 -1     1      1
```

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

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

```
    x      y  width height
    0      0     79     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 non-trivial GUI will require a window containing multiple widgets. Let us consider the case where the child of the window is itself a container, with multiple children. Essentially the same negotiation process occurs between the container and its children (the grandchildren of the window). The container calculates its size request based on the requests of its children and communicates it to the window. The size allocated to the container is then distributed to the children according to its layout algorithm. This process is the same for every level in the container hierarchy.

Box containers

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

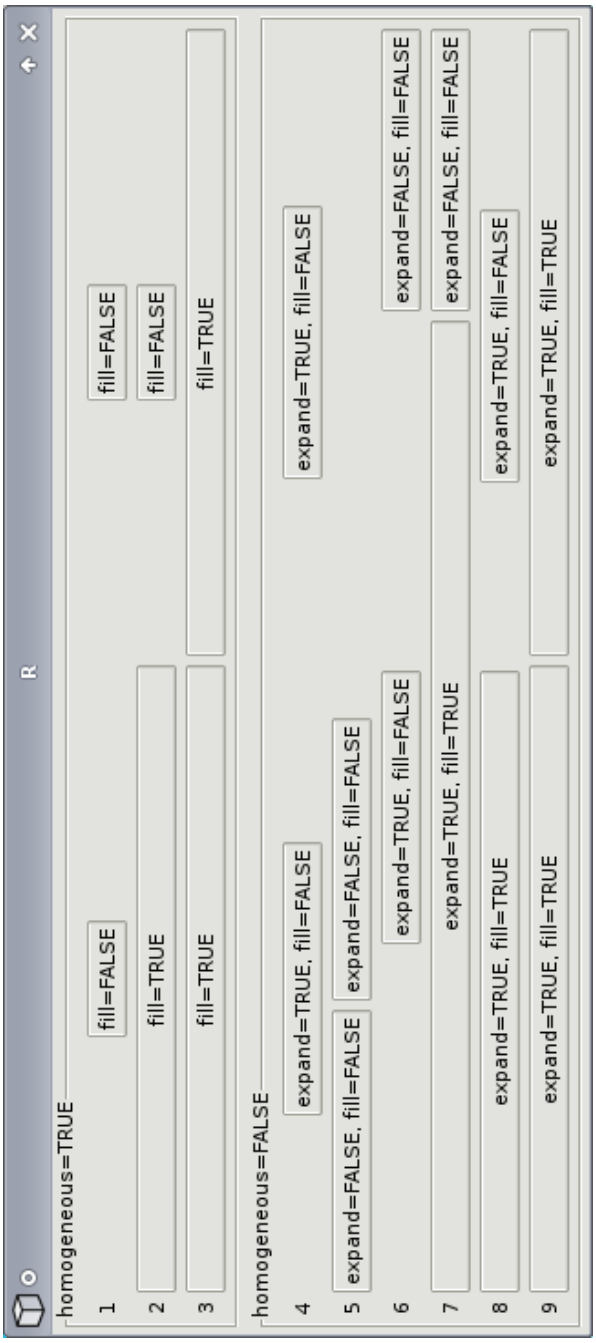

Figure 7.1: A screenshot demonstrating the effect of packing two buttons into `GtkHBox` instances using the `packStart` method with different combinations of the `expand` and `fill` settings. The effect of the `homogeneous` spacing setting on the `GtkHBox` is also shown.

neous. This means that the horizontal allocation of the box will be evenly distributed between the children. The second parameter directs the box to leave 5 pixels of space between each child. The following code constructs the GtkHBox:

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

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

The packStart and packEnd methods pack a widget into a box 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:

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

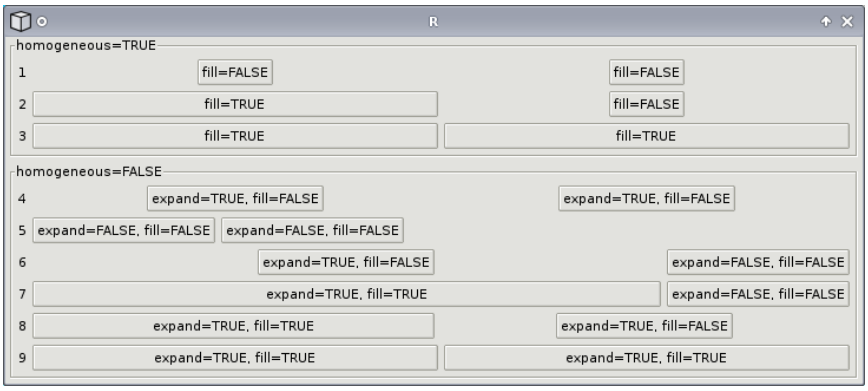

Figure 7.2: Examples of packing widgets into a box container. The top row shows no padding, whereas the 2nd and 3rd illustrate the difference between `padding` (an amount around each child) and `spacing` (an amount between each child). The last two rows show the effect of `fill` when `expand=TRUE`. This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

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

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

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

There are several ways to add space around widgets in a box container. The `spacing` argument for the constructors specifies the amount of space, in pixels, between the cells. This defaults to zero. The `pack` methods have a `padding` argument, also defaulting to zero, for specifying the padding in pixels on either side of the child. It is important to note the difference: `spacing` is between children and the same for every boundary, while the `padding` is specific to a particular child and occurs on either side, even on the ends. The spacing between widgets is the sum of the `spacing` value and the two `padding` values when the children are added. Example 8.3 provides an example and Figure 7.2 an illustration.

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 `g` to the second position:

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

Alignment

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

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

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

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

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

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

7.3 Dialogs

GTK+ provides a number of convenient dialogs for the common use cases, as well as 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:

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

The `flags` argument allows one 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 4 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 <- dlg$run()
if(response == GtkResponseType["cancel"] || # for other buttons
   response == GtkResponseType["close"] ||
   response == GtkResponseType["delete-event"]) {
  ## pass
} else if(response == GtkResponseType["ok"]) {
  print("Ok")
}
```

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

```
dlg$Destroy()
```

The return value can then be inspected for the action, such as what 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:

```
dlg <- 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, which is an instance of GtkVBox. To complete our dialog, we place a labeled text entry into the content area:

```
hb <- gtkHBox()
hb['spacing'] <- 10
#
hb$packStart(gtkLabel("Enter a value:"))
entry <- gtkEntry()
hb$packStart(entry)
#
vb <- dlg$getContentArea()
vb$packStart(hb)
```

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:

```
ID <- gSignalConnect(dlg, "response",
                     f=function(dlg, resp, user.data) {
                       if(resp == GtkResponseType["ok"])
                         print(entry$getText()) # Replace this
                       dlg$Destroy()
                     })
dlg$showAll()
dlg$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 may 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 the use of the dialog, the most commonly used of the three. An open file dialog can be created with:

```
dlg <- gtkFileChooserDialog(title="Open a file",
                            parent=NULL, action="open",
                            "gtk-ok", GtkResponseType["ok"],
                            "gtk-cancel", GtkResponseType["cancel"],
                            show=FALSE)
```

The dialog constructor allows one 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(dlg, "response", f=function(dlg, resp, data) {
  if(resp == GtkResponseType["ok"]) {
    filename <- dlg$getFilename()
    print(filename)
  }
  dlg$destroy()
})
```

The file selected is returned by `getFilename`. If multiple selection is enabled (via the `select-multiple` property) one should call the plural `getFilenames`.

For the open dialog, one 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$addPattern("*.Rdata")
dlg$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 filename 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 **??** for an example of printing R graphics using `cairoDevice`.

7.4 Special-purpose Containers

In Section **??**, we presented `GtkBox` and `GtkAlignment`, the two most useful layout containers in GTK+. This section introduces some other important containers. These include the merely decorative `GtkFrame`; the interactive `GtkExpander`, `GtkPaned` and `GtkNotebook`; and the grid-style layout container `GtkTable`. All of these widgets are derived from `GtkContainer`, and so share 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. `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:

```
nb <- gtkNotebook()
nb$appendPage(gtkLabel("Page 1"), gtkLabel("Tab 1"))
```

```
[1] 0
```

```
nb$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 allows for more complicated tabs, such as a box container with a label and 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:

```
nb['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:

```
nb['page'] <- 1
nb['page']
```

```
[1] 1
```

To move sequentially through the pages, calll the methods `nextPage` and `prevPage`. When the current page changes, the `switch-page` signal is emitted.

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`roperty. Pages can be deleted using the method `removePage`.

Managing Many Pages By default, a notebook will request enough space to display all of its tabs. If there are many tabs, space may be wasted. `GtkNotebook` solves this with the scrolling idiom. If the property `scrollable` is set to `TRUE`, arrows will be added to allow the user to scroll through the tabs. In this case, the tabs may become difficult to navigate. Setting the `enable-popup` property to `TRUE` enables a right-click popup menu listing all of the tabs for direct navigation.

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) {
    label <- gtkHBox()
    label$packStart(gtkLabel(label.text))
    icon <- gtkImage(pixbuf =
            object$renderIcon("gtk-close", "button", size="menu"))
    closeButton <- gtkButton()
    closeButton$setImage(icon)
    closeButton$setRelief("none")
    label$packEnd(closeButton)
    gSignalConnect(closeButton, "clicked", function(b) {
      index <- nb$pageNum(child)
      nb$removePage(index)
    })
    object$insertPage(child, label, position)
  }
```

Here is a simple demonstration of its usage:

```
w <- gtkWindow()
```

```
nb <- gtkNotebook(); w$add(nb)
nb$insertPageWithCloseButton(gtkButton("hello"),
                             label.text="page 1")
nb$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 scrollbars for the user to adjust the visible region of a single child. The range, step and position of GtkScrollbar are controlled by an instance of GtkAdjustment, just as with the slider and spin button. Scrolled windows are most often used with potentialy 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)
```

```
[1] TRUE
```

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$height * x)
  updateAdjustment <- function(adj) {
    adj$setValue(x * adj$getValue() + (x - 1) * adj$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 will 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 to 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(x, 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 scrollbars are drawn. By default, they are always drawn. The `"automatic"` value from the `GtkPolicyType` enumeration draws the scrollbars 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 contain two widgets, arranged horizontally or vertically and separated by a handle. The user may adjust the position of the handle to apportion the allocation between the widgets. 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 may 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)"))
```

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 gutter 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 form layout, which we will demonstrate in our example.

Example 7.3: Dialog layout
This example shows how to layout 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 3 rows and 2 columns:

```
tbl <- 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:

```
sizeLabel <- gtkLabel("Sample size:")
sizeCombo <- gtkComboBoxNewText()
```

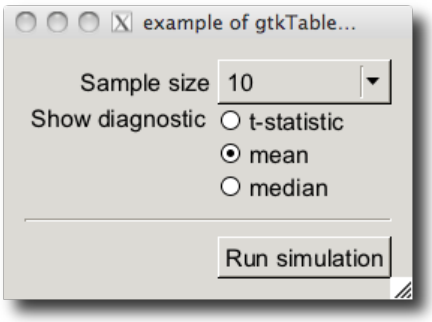

Figure 7.3: A basic dialog using a `gtkTable` container for layout.

```
sapply(c(5, 10, 15, 30), sizeCombo$appendText)
diagLabel <- gtkLabel("Diagnostic:")
diagRadios <- gtkVBox()
rb <- list()
rb$t <- gtkRadioButton(label="t-statistic")
rb$mean <- gtkRadioButton(rb, label="mean")
rb$median <- gtkRadioButton(rb, label="median")
sapply(rb, diagRadios$packStart)
submitBox <- gtkVBox()
submitBox$packEnd(gtkButton("Run simulation"), expand = FALSE)
```

The labels need to be aligned to the right, up against their corresponding entry widgets, which should be left-aligned:

```
sizeLabel['xalign'] <- 1
diagLabel['xalign'] <- 1; diagLabel['yalign'] <- 0
diagAlign <- gtkAlignment(xalign = 0)
diagAlign$add(diagRadios)
```

The labels are aligned through the `GtkMisc` functionality inherited by `GtkLabel`. The `GtkVBox` with the radio buttons does not support this, so we need to embed it within a `GtkAlignment`. We have aligned the diagnostic label to the top of its cell; otherwise, it would have been in the middle. The radio buttons are left aligned, up against the label.

Child widgets are added to a table through the `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 may be overridden by specifying additional arguments to `attach`.

The following attaches the combo box, radio buttons and their labels to the table:

```
tbl$attach(sizeLabel, left.attach=0,1, top.attach=0,1,
           xoptions = c("expand", "fill"), yoptions="")
tbl$attach(sizeCombo, left.attach=1,2, top.attach=0,1,
           xoptions="fill", yoptions="")
#
tbl$attach(diagLabel, left.attach=0,1, top.attach=1,2,
           xoptions = c("expand", "fill"),
           yoptions=c("expand", "fill"))
#
tbl$attach(diagAlign, left.attach=1,2, top.attach=1,2,
           xoptions=c("expand", "fill"), yoptions = "")
#
tbl$attach(submitBox, 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.

One can add spacing to the right of cells in a particular row or column. Here we add 5 pixels of space to the right of the label column:

```
tbl$setColSpacing(0, 5)
```

We complete the example by placing the table into a window:

```
w <- gtkWindow(show=FALSE)
w['border-width'] <- 14
w$setTitle("GtkTable Example")
w$add(tbl)
```

8

RGtk2: Basic Components

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

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

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

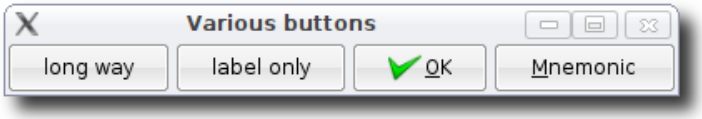

Figure 8.1: Various buttons

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 checkmark) 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`

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

```
b$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 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.

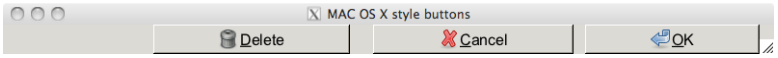

Figure 8.2: Example using stock buttons with extra spacing added between the `delete` and `cancel` buttons.

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 12 pixels, while potentially dangerous buttons are separated from the rest by 24 pixels, as per the Mac 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.

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

The padding occurs to the left and right of the child. The `ok` button is given no padding. The `cancel` button is packed with 12 pixels of spacing, which separates it from the `ok` button. Recognizing the `delete` button as potentially irreversible, we add 12 pixels of separation between it and the `cancel` button, for a total of 24 pixels. The blank label pushes the buttons against the right side of the box. We instruct the `ok` button to grab focus, so that it becomes the default button:

```
ok$grabFocus()
```

Figure 8.3: Various formatting for a label: wrapping, alignment, ellipsizing, PANGO markup

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 being 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$setMarkup(sprintf(tmpl, string))
#
sapply(list(basicLabel, ellipsized, rightJustified, pangoLabel),
       g$packStart, expand = TRUE, fill = TRUE)
w$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.

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 `<b>` for bold, `<i>` for italics, `<ul>` 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 `<span>` tag markup, such as `<span color='red'>some text</span>`. 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 interactive, one 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, images are often a more space efficient alternative to labels. `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.

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

```
[1] TRUE
```

```
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 one wishes 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 are returned in a list by `gtkStockListIds`. The first 3 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 stock identifier.

8.3 Input Controls

Text entry

The widgets explained thus far are largely static. For example, GTK+ does not yet support editable labels. GTK+ has two different widgets for editing text. One is optimized for multi-line text documents, the other for single line entry. We will discuss complex multi-line text editing in Section 9.6. For entering a single line of text, the `GtkEntry` widget is appropriate:

```
e <- 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. We connect to the signal and obtain the entered text upon activation:

```
gSignalConnect(e, "activate", function() {
  message("Text entered: ", e$getText())
})
```

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. Instead, one should call `setMaxLength`.

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 example shows how to insert and then delete text:

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

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

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

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

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

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.

One can set an icon on an entry from a `GdkPixbuf`, stock ID, icon name, or `GIcon`. Two icons are possible, one at the beginning (`primary`) and one at the end (`secondary`). For example, 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")
    validatedEntry$setIconTooltipText("primary",
                                      "Only letters are allowed")
  }
  else {
    entry$setIconFromStock("primary", "gtk-yes")
```

```
    validatedEntry$setIconTooltipText("primary", NULL)
  }
})
validatedEntry$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/logical) and is usually proxied by a `GtkCheckButton`.

A `GtkCheckButton` is constructed by `gtkCheckButton`:

```
cb <- gtkCheckButton("Option")
```

The state of the binary variable is represented by the `active` property. We check our button:

```
cb['active']
```

```
[1] FALSE
```

```
cb['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(cb, "toggled", function(x) {
  message("Button is ", if (x$active) "active" else "inactive")
})
```

An alternative to `GtkCheckButton` is the lesser used `GtkToggleButton`, which is actually the parent class of `GtkCheckButton`. A toggle button is drawn as an ordinary button. It remains depressed while the state variable is `TRUE`, instead of relying on a check box 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 `sapply` idiom that avoids the need to allocate a list and populate it in a `for` loop.

We add each button to a vertical box:

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

We can set and query which button is active:

```
g[[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(w, data) {
         if(w['active']) # set before callback
           message("clicked", w$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)
w <- gtkWindow()
w['title'] <- "Radio group example"
g <- gtkVBox(); w$add(g)
sapply(rev(radiogp$getGroup()), gtkBoxPackStart, object = g)
```

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 radio buttons and is better suited for 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 construct and populate a simple combo box:

```
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:

```
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(w, ...) {
                 if(w$getActive() < 0)
                   cat("No value selected\n")
                 else
                   cat("Value is", w$getActiveText(), "\n")
               })
```

Although combo boxes are much more space efficient than radio buttons, it can be difficult to use a combo box when there are a large number of items. The `setWrapWidth` 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:

```
w <- gtkWindow(show=FALSE)
w$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.

```
g <- gtkVBox(); w$add(g)
#
g1 <- gtkHBox(); g$packStart(g1)
g1$packStart(gtkLabel("Data frames:"))
g1$packStart(df_combo)
#
g2 <- gtkHBox(); g$packStart(g2)
g2$packStart(gtkLabel("Variable:"))
g2$packStart(var_combo)
g2$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(w, ...) {
  var_combo$getModel()$clear()
  sapply(find_vars(w$getActiveText()),  var_combo$appendText)
  g2$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

The slider widget and spin button widget allow selection from a regularly spaced, semi-continuous list of values. `GtkScale` implements a slider and may be oriented either horizontally or vertically. This depends on the class: `GtkHScale` or `GtkVScale`.

Example 8.9: A slider controlling histogram bin selection
We demonstrate a slider for controlling the bin size of a histogram. First, we create and configure the horizontal slider:

```
slider <- gtkHScale(min = 1, max = 100, step = 1)
slider$setValue(10)
slider['value-pos'] <- "bottom"
```

We specify the minimum, maximum and step values for the scale. This set of values is formally represented by the `GtkAdjustment` structure, which could serve as a data model for synchronizing multiple sliders or other scale-based widgets. Ordinarily, it is not necessary to construct a `GtkAdjustment` explicitly. Instead, one passes the values as parameters to the constructor. The initial value is set to 10, and the value will be rendered in a label beneath the slider, according to the `value-pos` property.

The `value-changed` signal is emitted whenever the slider is adjusted. Our handler updates the histogram:

```
gSignalConnect(slider, "value-changed",
               f = function(w, ...) {
                 val <- w$getValue()
                 drawHistogram(val)
               })
```

Finally, we load the data, define the `drawHistogram` function, and finish the GUI:

```
data <- rnorm(100)
library(lattice)
drawHistogram <- function(val) print(histogram(data, nint = val))
drawHistogram(slider$getValue())
w <- gtkWindow(); w$setTitle("Histogram bin selection")
w$add(slider)
```

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`, demonstrated above, specifies where this value will be drawn using values from `GtkPositionType`. The default is `top`.

Spin buttons

The spin button widget is very similar to the slider widget, conceptually and in terms of the GTK+ API. Spin buttons are constructed with `gtkSpinButton`. As with sliders, this constructor requires specifying adjustment values, either as a `GtkAdjustment` or individually. The methods `getValue` and `setValue` once again get and set the value. The `value-changed` signal is emitted when the spin button value is changed.

Example 8.10: A range widget
This example shows how to make a range widget that combines both the slider and spinbutton to choose a single number. Such a widget is 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.

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.

```
g <- gtkHBox()
g$packStart(slider, expand=TRUE, fill=TRUE, padding=5)
g$packStart(spinbutton, expand=FALSE, padding=5)
```

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.

8.4 Progress Reporting

Progress bars

It is common to use a progress bar to indicate the progress of a long running computation. This implemented by `GtkProgressBar`. A text label describes the current operation, and the progress bar communicates the fraction completed:

```
w <- gtkWindow(); w$setTitle("Progress bar example")
pb <- gtkProgressBar()
w$add(pb)
#
pb$setText("Please be patient...")
for(i in 1:100) {
  pb$setFraction(i/100)
  Sys.sleep(0.05) ## replace with a step in the process
}
pb$setText("All done.")
```

Progress bars can also show indefinite activity by periodically pulsing the bar:

```
pb$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 basic setup is one adds pages to the assistant object and they are navigated in a linear manner. In our example, we will see how to 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 may 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 are added to the assistant through the `appendPage`, `insertPage`, or `prependPage` methods. Pages are referred to by the `GtkWidget` object or 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. This is needed to create dynamically generated pages.

Example 8.11: 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.

We begin by defining our assistant and connecting to its `cancel` signal.

```
asst <- gtkAssistant(show=FALSE)
asst$setSizeRequest(500, 500)
```

```
gSignalConnect(asst, "cancel", function(asst) asst$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=asst)
sapply(pages, gtkAssistantSetPageType, object=asst,
       type=page_types)
```

We customize each page with a logo, using a side log here.

```
image <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
sapply(pages, gtkAssistantSetPageSideImage, object=asst,
       pixbuf=image)
```

When a page is about to be called we check and see if it has any children, if not we call a function to create the page. These functions are stored in a list so that they can be called by page index.

```
populatePage <- list()
gSignalConnect(asst, "prepare", function(a, w, data) {
  page_no <- which(sapply(pages, identical, w))
  if(!length(w$getChildren()))
    populatePage[[page_no]]()
})
```

Although we don't 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. This function simply returns an integer with the next page number based on the last one.

```
asst$setForwardPageFunc(function(i, data) {
  ifelse(i == 0 && have_CRAN(), 2L, as.integer(i + 1))
}, data=NULL)
```

```
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 a welcome and simply shows a label. It is immediately complete.

```
populatePage[[1]] <- function() {
  asst$setPageTitle(pages[[1]], "Install a CRAN package")
  pages[[1]]$packStart(l <- gtkLabel())
  pages[[1]]$packStart(gtkLabel(), expand=TRUE) # a spring
```

```
  l$setMarkup(paste(
       "<span font='x-large'>Install a CRAN package</span>",
       "This wizard will help install a package from <b>CRAN</b>",
       "If you have not already specified a CRAN repository, one",
       "you will be prompted to do so.",
       sep="\n"))
  asst$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 is a summary of the package taken from CRAN and a chance for the user to configure a few options for `install.packages`.

```
populatePage[[4]] <- function() {
  asst$setPageTitle(pages[[4]], "Install CRAN package")
  #
  get_desc <- function(pkgname) {
    tpl <- "http://cran.r-project.org/web/packages/%s/DESCRIPTION"
    x <- readLines(sprintf(tpl, pkgname))
    f <- tempfile(); cat(paste(x, collapse="\n"), file=f)
    read.dcf(f)
  }
  pkg_desc <- get_desc(CRAN_package)
  #
  l <- gtkLabel()
  l$setMarkup(paste(
    sprintf("Install package: <b>%s</b>", pkg_desc[1, 'Package']),
    "\n",
    sprintf("%s", pkg_desc[1, 'Description']),
    sep="\n"))

  pages[[4]]$packStart(l)
  #
  tbl <- gtkTable()
  pages[[4]]$packStart(tbl, 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(w, ...) {
    install_options['type']] <<- pkg_types[1 + w$getActive()]
  })
```

```
  tbl$attachDefaults(gtkLabel("Package type:"), 0, 1, 0, 1)
  tbl$attachDefaults(combo, 1, 2, 0, 1)

  #
  cb <- gtkCheckButton()
  cb$setActive(TRUE)
  gSignalConnect(cb, "toggled", function(w) {
    install_options[['dependencies']] <<- w$getActive()
  })
  tbl$attachDefaults(gtkLabel("Install dependencies"), 0, 1, 1, 2)
  tbl$attachDefaults(cb, 1, 2, 1, 2)

  #
  fc <- gtkFileChooserButton("Select a directory...",
                             "select-folder")
  fc$setFilename(.libPaths()[1])
  gSignalConnect(fc, "selection-changed", function(w) {
    dir <- w$getFilename()
    install_options[['lib']] <<- dir
  })
  tbl$attachDefaults(gtkLabel("Where"), 0, 1, 2, 3)
  tbl$attachDefaults(fc, 1, 2, 2, 3)

  asst$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 real application would reimplement that. Instead we just set a message once the package install attempt is done.

```
populatePage[[5]] <- function() {
  asst$setPageTitle(pages[[5]], "Done")
  install_options$pkgs <- CRAN_package
  out <- try(do.call("install.packages", install_options),
             silent=TRUE)

  l <- gtkLabel(); pages[[5]]$packStart(l)
  if(!inherits(out, "try-error")) {
    l$setMarkup(sprintf("Package %s installed successfully",
                        CRAN_package))
  } else {
    l$setMarkup(paste(sprintf("Package %s failed to install",
                              CRAN_package),
                      paste(out, collapse="\n"),
                      sep="\n"))
  }
```

```
  asst$setPageComplete(pages[[5]], FALSE)
}
```

To finish we simply need to populate the first page and call the assistant's `show` method.

```
populatePage[[1]]()
asst$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)
```

```
[1] TRUE
```

```
win <- gtkWindow(show=FALSE)
win$add(device)
win$showAll()
plot(mpg ~ hp, data = mtcars)
```

We create the `GtkDrawingArea`, coerce it to a Cairo-based graphics device, and place it in a window. Finally, we display a scatterplot. 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 8.12: 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`:

```
printOp <- 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(printOp, "draw-page", function(x, 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:

```
printOp$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.

8.7 Drag and drop

A drag and drop operation is the movement of data from a source widget to a target widget. 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
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 only use the `"text"` target in this example.

We construct a button and call `gtkDragSourceSet` to instruct it to act as a drag source:

```
w <- gtkWindow(); w['title'] <- "Drag Source"
dragSourceWidget <-  gtkButton("Text to drag")
w$add(dragSourceWidget)
gtkDragSourceSet(dragSourceWidget,
       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(dragSourceWidget, "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:

```
w <- gtkWindow(); w['title'] <- "Drop Target"
dropTargetWidget <- gtkButton("Drop here")
w$add(dropTargetWidget)
gtkDragDestSet(dropTargetWidget,
               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 only need 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`nd performing some operation with it. In this case, we set the text on the button:

```
gSignalConnect(dropTargetWidget, "drag-data-received",
       function(widget, context, x, y, sel, tType, eTime) {
         dropdata <- sel$getText()
         widget$setLabel(rawToChar(dropdata))
       })
```

The `context`rgument is a `GdkDragContext`, containing information about the drag event. The `x`nd `y`rguments 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`.

9

RGtk2: Widgets Using Data Models

Many widgets in GTK+ use the model, view, controller (MVC) paradigm. For most, like the button, the MVC pattern is implicit; however, widgets that primarily display data explicitly incorporate the MVC pattern into their design. The data model is factored out as a separate object, while the widget plays the role of the view and controller. The MVC approach adds a layer of complexity but facilitates the display of the dynamic data in multiple, coordinated views.

9.1 Display of tabular data

Widgets that display lists, tables and trees are all based on the same basic data model, `GtkTreeModel`. Although its name suggests a hierarchical structure, `GtkTreeModel` is also tabular. We first describe the display of an R data frame in a list or table view. The display of hierarchical data, as well as further details of the `GtkTreeModel` framework, are treated subsequently.

Loading a data frame

As an interface, `GtkTreeModel` may be implemented in any number of ways. GTK+ provides simple in-memory implementations for hierarchical and non-hierarchical data. For improved speed, convenience and familiarity, `RGtk2` includes a custom `GtkTreeModel` implementation called `RGtkDataFrame`, which is based on an R data frame. For non-hierarchical data, this is usually the model of choice, so we discuss it first.

R uses data frames to hold tabular data, where each column is of a certain class, and each row is related to some observational unit. This fits the structure of `GtkTreeModel` when there is no hierarchy. As such it is natural to have a means to map a data frame into a store for a tree view. `RGtkDataFrame` implements `GtkTreeModel` to perform this role and is constructed with the `rGtkDataFrame` function. Populating a `RGtkDataFrame` is 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. The constructor takes a data frame as an argument. The column classes are important, so even if this data frame is empty, the user should specify the desired column classes upon construction.

An object of class `RGtkDataFrame` supports the familiar S3 methods `[`, `[<-`, `dim`, and `as.data.frame`. The `[<-` method does not have quite the same functionality as it does for a data frame. Columns can not be removed by assigning values to `NULL`, and column types should not be changed. These limitations are inherit in the design of GTK+: columns may not be removed from `GtkTreeModel`, and views expect the data type to remain the same.

Example 9.1: Defining and manipulating a `RGtkDataFrame`
The basic data frame methods are similar.

```
data(Cars93, package="MASS")              # mix of classes
model <- rGtkDataFrame(Cars93)
model[1, 4] <- 12
model[1, 4]                               # get value
```

```
[1] 12
```

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, one should add rows with `appendRows` and add columns with `appendColumns` (or sub-assignment, `[<-`).

The `setFrame` method replaces the underlying data frame.

```
model$setFrame(Cars93[1:5, 1:5])
```

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 tree view is essentially a container of columns, where every column has the same number of rows. If the view has a single column, it is essentially a list. If there are multiple columns,

it is a table. If the rows are nested, it is a tree table, where every node has values on the same columns.

A tree view is constructed by `gtkTreeView`:

```
view <- gtkTreeView(model)
```

Usually, as in the above, the model is passed to the constructor. Otherwise, the model may be accessed with `setModel` and `getModel`.

A newly created tree view displays zero columns, regardless of the number of columns in the model. Each column, an instance of `GtkTreeViewColumn`, must be constructed, inserted into the view and instructed to render content based on one or more columns in the data model:

```
vc <- gtkTreeViewColumn()
vc$setTitle("Manufacturer")
cr <- gtkCellRendererText()
vc$packStart(cr)
vc$addAttribute(cr, "text", 0)
view$insertColumn(vc, 0)
```

A column with the title "Manufacturer" is inserted at the first, 0-based, position. For displaying a simple data frame, we only need to render text. Each row in a column consists of one or more cells, managed in a layout. The number of cells and how each cell is rendered is uniform down a column. As an implementation of `GtkCellLayout`, `GtkTreeViewColumn` delegates the responsibility of rendering to one or more `GtkCellRenderer` objects. The cell renderers are packed into the column, which behaves much like a box container. Rendering of text cells is the role of `GtkCellRendererText`; we create an instance with `gtkCellRendererText`. There are several properties that control how the text is rendered. A so-called *attribute* links a model column to a renderer property. The most important property is `text`, the text itself. In the example, we bind the `text` property to the first (0-indexed) column in the model.

`GtkTreeView` provides the `insertColumnWithAttributes` convenience method to perform all of these steps with a single call. We invoke it to add a second column in our view:

```
view$insertColumnWithAttributes(position = -1,
     title = "Model", cell = gtkCellRendererText(), text = 1)
```

The -1 passed as the first argument indicates that the column should be appended. Next, we specify the column title, a cell renderer, and an attribute that links the `text` renderer property to the second column in the model. In general, any number of attributes may be defined after the third argument. We will use the above idiom in all of the following examples, as it is much more concise than performing each step separately.

To display the entire Cars93 data frame, we insert a view column for every column in the data frame. Here, we reconstruct the view, inserting a view column for every column in the data frame, i.e., the model.

```
view <- gtkTreeView(model)
mapply(view$insertColumnWithAttributes, -1, colnames(model),
       list(gtkCellRendererText()),
       text = seq_len(ncol(model)) - 1)
```

Although it was relatively easy to create a `GtkTreeModel` for the data frame using `RGtkDataFrame`, the complexity of `GtkTreeView` complicates the task of displaying the data frame in a simple, textual table. When this is all that is necessary, one might consider `gtable` from `gWidgets`. For those who wish to render text in each row differently (e.g., in a different color) or fill cells with images, check boxes, progress bars and the like, direct use of the `GtkTreeView` API is required.

Manipulating view columns The `GtkTreeView` widget is essentially a collection of columns. Columns are added to the tree view with the methods `insertColumn` or, as shown above, `insertColumnWithAttributes`. A column can be moved with the `moveColumnAfter` method, and removed with the `removeColumn` method. The `getColumns` method returns a list containing all of the tree view columns.

There are several properties for controlling the behavior and dimensions of a `GtkTreeViewColumn` instance. The property `"resizable"` determines whether the user can resize a column, by dragging with the mouse. The size properties `"width"`, `"min-width"`, and `"fixed-width"` control the size. The visibility of the column can be adjusted through the `setVisible` method.

Additional Features Tree views have several special features, including sorting, incremental search and drag-n-drop reordering. Sorting is discussed in Section 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 tree view will popup a search box when the user types control-f. To designate an arbitrary text entry widget as the search box, call `setSearchEntry`. The entry can be placed anywhere in the GUI. Columns are always reorderable by drag and drop. Reordering rows through drag-and-drop is enabled by the `reorderable` property.

Aesthetic properties `GtkTreeView` is capable of rendering some visual guides. The `rules-hint`, if `TRUE`, will instruct the theme to draw rows in alternating colors. To show grid lines, set `enable-grid-lines` to `TRUE`.

Accessing GtkTreeModel

Although RGtkDataFrame provides a familiar interface for manipulating the data in a GtkTreeModel, it is often necessary to directly interact with the GTK+ API, such as when using another type of data model or interpreting user selections. There are two primary ways to index into the rows of a tree model: paths and iterators.

To index directly into an arbitrary row, a GtkTreePath is appropriate. For a 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 may 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:

```
secondRow <- gtkTreePathNewFromIndices(2)
```

Referring to a row in a hierarchy is slightly more complex:

```
abcPath <- gtkTreePathNewFromIndices(c(1, 3, 2))
abcPath <- gtkTreePathNewFromString("1:3:2")
```

In the above, both paths refer to the second child of the third child of the first top-level node. To recover the integer or string representation of the path, use getIndices or toString, respectively.

The second means of row indexing is through an iterator, GtkTreeIter, which is better suited for traversing a model. While a tree path is an intuitive, transparent row index, an iterator 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. The iterator design also prevents the use of the apply functions, which are generally preferred over the `while` loop for reasons of performance and clarity. An improvement would be to obtain the number of children, generate the sequence of row indices and access the row for each index:

```
nrows <- model$iterNChildren(NULL)
manufacturer <- sapply(seq(nrows), function(i) {
  iter <- model$iterNthChild(NULL, i)
  model$get(iter$iter, 0)[[1]]
})
```

Here we use `NULL` to refer to the virtual root node that sits above the rows in our table. Unfortunately, this usage too is unintuitive and slow, so the benefits of `RGtkDataFrame` should be obvious.

One can convert between 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, use a `GtkTreeRowReference`.

Selection

There are multiple modes of user interaction with a tree view: if the cells are not editable, then selection is the primary mode. A single click selects the value, and a double click is often used to initiate an action. If the cells are editable, then a double click or a click on an already selected row will initiate editing of the content. Editing of cell values is a complex topic and is handled by derivatives of `GtkCellRenderer`, see Section 9.1. Here, we limit our discussion to selection of rows.

GTK+ provides the class `GtkTreeSelection` to manage row selection. Every tree view has a single instance of `GtkTreeSelection`, returned by the `getSelection` method.

The usage of the selection object depends on the selection mode, i.e., whether multiple rows may be selected. The mode is configured with the

setMode method, with values from GtkSelectionModel, 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.

```
[1] 1
```

If our tree view is shown and a selection made, this code will return the value in the first column:

```
curSel <- selection$getSelected()
with(curSel, model$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) {
  curSel <- selection$getSelectedRows()
  if(length(curSel$retval)) {
    rows <- sapply(curSel$retval, gtkTreePathGetIndices) + 1L
    curSel$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 tree view. The callback has arguments tree.view pointing to the widget that emits the signal, path storing a tree path of the selected row, and column containing the tree view column. The column number is not returned. If that is of interest, it can be passed in via the user data argument, or matched against the children of the tree view through a command like

```
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 only necessary to inform each view column of the model column to use for sorting when the header is clicked:

```
vc <- view$getColumn(0)
vc$setSortColumnId(0)
```

In the above, clicking on the header of the first view column, `vc`, will sort by the first model column. Behind the scenes, `GtkTreeViewColumn` will set its sort column as the sort column on the model, i.e.:

```
model$setSortColumnId(0, "ascending")
```

Some models, however, do not implement `GtkTreeSortable`, such as `GtkTreeModelFilter`, introduced in the next section. Also, sorting a model permanently changes the order of its rows, which may be undesirable in some cases. The solution is to proxy the original model with a sortable model. The proxy obtains all of its data from the original model and reorders the rows according to the order of the sort column. GTK+ provides `GtkTreeModelSort` for this:

```
store <- rGtkDataFrame(mtcars)
sorted <- gtkTreeModelSortNewWithModel(store)
view <- gtkTreeView(sorted)
view$insertColumnWithAttributes(0, "Click to sort",
                                gtkCellRendererText(), text=0)
view$getColumn(0)$setSortColumnId(0)
```

When the user sorts the table, the underlying `store` will not be modified.

The default sorting function can be changed by calling the method `setSortFunc` on a sortable model. The following function shows how the default sorting might be implemented.

```
f <- function(model, iter1, iter2, user.data) {
  column <- user.data
  val1 <- model$getValue(iter1, column)$value
  val2 <- model$getValue(iter2, column)$value
  as.integer(val1 - val2)
}
sorted$setSortFunc(sort.column.id=0, sort.func=f)
```

Filtering

The previous section introduced the concept of a proxy model in `GtkTreeModelSort`. Another common application of proxying is filtering. For filtering via a proxy model, GTK+ provides the `GtkTreeModelFilter` class. The basic idea is that an extra column in the base model stores logical values to indicate if a row should be visible. The index of that column is passed to the filter model, which provides only those rows where the filter column is `TRUE`.

This is the basic usage:

```
df <- data.frame(col=letters[1:3], vis=c(TRUE, TRUE, FALSE))
store <- rGtkDataFrame(df)
filtered <- store$filter()
filtered$setVisibleColumn(1)            # 0-based
view <- gtkTreeView(filtered)
```

The constructor of the filter model is `gtkTreeModelFilter`, which, somewhat coincidentally, also works as a method on the base model, i.e., `model$filter()`. To retrieve the original model from the filter, call its `getModel` method. The method `setVisibleColumn` specifies which column in the model holds the logical values. To customize filtering, one 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 may 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))
store <- rGtkDataFrame(df)
```

The filtered store needs to have the column specified that contains the logical values; in this example, it is the last column.

```
filteredStore <- store$filter()
filteredStore$setVisibleColumn(ncol(df)-1)      # offset
view <- gtkTreeView(filteredStore)
```

Next, we create a basic view of a single column:

```
view$insertColumnWithAttributes(0, "Col",
                 gtkCellRendererText(), text = 0)
```

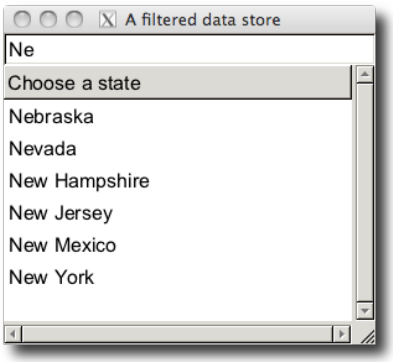

Figure 9.1: Example of a data store filtered by values typed into a text-entry widget.

An entry widget will be used to control the filtering. In the callback, we adjust the `visible` column of the `rGtkDataFrame` instance to reflect the rows to be shown. When `val` is an empty string, the result of `grepl` is `TRUE`, so all rows will be shown.

```
e <- gtkEntry()
gSignalConnect(e, "changed", function(w, data) {
  pattern <- w$getText()
  df <- data$getModel()
  values <- df[, "state.name"]
  df[, "visible"] <- grepl(pattern, values)
}, data=filteredStore)
```

Figure 9.1 shows the two widgets placed within a simple GUI.

Cell renderer details

The values in a tree model are rendered in a rectangular cell by the derivatives of `GtkCellRenderer`. Cell renderers are interactive, in that they also manage editing and activation of cells.

A cell renderer is independent of any data model. Its rendering role is limited to drawing into a specified rectangular region according to its current property values. An object that implements the `GtkCellLayout` interface, like `GtkTreeViewColumn` and `GtkComboBox` (see Section 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 visual-

ization 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:

```
renderer <- gtkCellRendererText()
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. There are several other attributes that can changed. For example, we display right-aligned text in a Helvetica font:

```
cr <- gtkCellRendererText()
cr['xalign'] <- 1                    # default 0.5 = centered
cr['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. The tree view object and the column index are not passed to the callback, unless one uses a closure or user data. For example, here is how one can update an `RGtkDataFrame` model from within the callback:

```
cr['editable'] <- TRUE
ID <- gSignalConnect(cr, "edited",
f=function(cr, path, newtext, user.data) {
  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, one 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, one must advance the cursor and activate editing of the next cell. For `GtkTreeView`, this is implemented by the `setCursor` method. The `path` argument takes a tree path instance, the `column` argument should be a tree view column object, and the flag `start.editing` indicates whether to initiate editing.

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

```
m <- rGtkDataFrame(df)
v <- gtkTreeView(m)
```

and configure the first column

```
cr <- gtkCellRendererText()
cr['background'] <- 'gray80'
v$insertColumnWithAttributes(position=-1,
                             title="Option",
                             cell=cr,
                             text=1 - 1)
```

```
[1] 1
```

The first column has a special background color which we specify 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 if a cell has been edited:

```
cr <- gtkCellRendererText()
cr['editable'] <- TRUE
v$insertColumnWithAttributes(position = -1,
                             title = "Value",
                             cell = cr,
                             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 one to easily find the values which were edited.

```
gSignalConnect(cr, "edited", function(cr, path, new.text,
                                      user.data) {
  m <- user.data$model
  i <- as.numeric(path) + 1; j <- user.data$column
  m[i,j] <- as(as(new.text, m[i, 'class']), "character")
  m[i, 'dirty'] <- TRUE                        # mark dirty
  m[i, 'edit_color'] <- 'gray70'               # change color
}, data=list(model=m, column=2))
```

A simple window displays our GUI.

```
w <- gtkWindow(show=FALSE)
w['title'] <- "Option editor"
w$setSizeRequest(300,500)
sw <- gtkScrolledWindow()
w$add(sw)
sw$add(v)
w$show()
```

Implementing this 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 the `helpr` function 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)
```

It 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. The purpose of this example is only to demonstrate tooltips on a tree view. For many widgets, adding a tooltip is as easy as calling `setTooltipText`. However, it is more complicated in a tree view, as each cell should get a different tip. To add tooltips to the tree view we first indicate that we want tooltips, then connect to the `query-tooltip` signal to implement the tooltip:

```
v["has-tooltip"] <- TRUE
gSignalConnect(v, "query-tooltip",
       function(w, x, y, key_mode, tooltip, user.data) {
         out <- w$getTooltipContext(x, y, key_mode)
         if(out$retval) {
           m <- w$getModel()
           i <- as.numeric(out$path$toString()) + 1
           val <- m[i, "option"]
           txt <- descs[[val]]$desc
           txt <- gsub("code>","b>", txt)  # no code in PANGO
           tooltip$setMarkup(txt)
           TRUE
         } else {
           FALSE                           # no tooltip
         }
       })
```

Within this callback we check 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 only uses 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.

```
cr <- gtkCellRendererCombo()
store <- rGtkDataFrame(state.name)
cr['model'] <- store
cr['text-column'] <- 0
cr['editable'] <- TRUE                    # needed
```

The spin button editor is configured by setting a `GtkAdjustment` on the `adjustment` property.

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`, one 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. A similar mechanism is part of the SPSS model specification GUI of Figure 1.3. 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`:

```
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`:

```
mdf <- data.frame(Variables = I(sort(names(df))),
                  icon = I(sapply(df, make_icon_pixmap)),
                  selected = rep(FALSE, ncol(df)))
model <- rGtkDataFrame(mdf)
```

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:

```
selectedFilter <- model$filter()
selectedFilter$setVisibleColumn(2)
unselectedFilter <- model$filter()
unselectedFilter$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:

```
unselectedView <- gtkTreeView(unselectedFilter)
selectedView <- gtkTreeView(selectedFilter)
unselectedView$getSelection()$setMode('multiple')
selectedView$getSelection()$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() {
  vc <- gtkTreeViewColumn()
  vc$setTitle("Variable")
  cr <- gtkCellRendererPixbuf()
  vc$packStart(cr)
  vc$addAttribute(cr, "pixbuf", 1)
  cr <- gtkCellRendererText()
  vc$packStart(cr)
  vc$addAttribute(cr, "text", 0)
  vc
}
unselectedView$insertColumn(make_view_column(), 0)
selectedView$insertColumn(make_view_column(), 0)
```

For later use we extend the API for a tree view – one method to find the selected indices (1-based) and one to indicate if there is a selection:

```
gtkTreeViewSelectedIndices <- function(object) {
  paths <- object$getSelection()$getSelectedRows()$retval
  out <- sapply(paths, function(i) {
    model <- object$getModel()          # Filtered!
    model$convertPathToChildPath(i)$toString()
  })
  if(length(out) == 0)
    integer(0)
  else
    as.numeric(out) + 1                              # 1-based
}
#
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:

```
selectButton <- gtkButton(">")
unselectButton <- gtkButton("<")
toggleSelectionOnClick <- function(button, view) {
  gSignalConnect(button, "clicked", function (x) {
    ind <- view$selectedIndices()
    model[ind, "selected"] <- !model[ind, "selected"]
  })
}
toggleSelectionOnClick(selectButton, unselectedView)
toggleSelectionOnClick(unselectButton, selectedView)
```

We only want our buttons sensitive if there is a possible move. This is determined by the presence of a selection:

```
selectButton['sensitive'] <- FALSE
unselectButton['sensitive'] <- FALSE
trackSelection <- function(button, view)
  gSignalConnect(view$getSelection(), "changed",
    function(x) button['sensitive'] <- view$hasSelection())
trackSelection(selectButton, unselectedView)
trackSelection(unselectButton, selectedView)
```

We now layout our GUI using a horizontal box, into which is packed the views and a box holding the selection buttons. The views will be scrollable, so we place them in scrolled windows:

```
w <- gtkWindow(show=FALSE)
w$setDefaultSize(600, 400)
g <- gtkHBox()
w$add(g)
selectedScroll <- gtkScrolledWindow()
```

```
selectedScroll$add(selectedView)
selectedScroll$setPolicy("automatic", "automatic")
unselectedScroll <- gtkScrolledWindow()
unselectedScroll$add(unselectedView)
unselectedScroll$setPolicy("automatic", "automatic")
buttonBox <- gtkVBox()
centeredBox <- gtkVBox()
buttonBox$packStart(centeredBox, expand = TRUE, fill = FALSE)
centeredBox$setSpacing(12)
centeredBox$packStart(selectButton, expand = FALSE)
centeredBox$packStart(unselectButton, expand = FALSE)
g$packStart(unselectedScroll, expand=TRUE)
g$packStart(buttonBox, expand=FALSE)
g$packStart(selectedScroll, expand=TRUE)
```

Finally, we show the top-level window:

```
w$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.

```
cr <- gtkCellRendererToggle()
cr['activatable'] <- TRUE             # cell can be activated
cr['active'] <- TRUE
gSignalConnect(cr, "toggled", function(w, path) {
  model$active[as.numeric(path) + 1] <- w['active']
})
```

To render the toggle as a radio button instead of a check box, 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 check box column in a tree view
This example demonstrates the construction of a GUI for selecting one or more rows from a data frame. We will display a list of the installed packages that can be upgraded from CRAN, although this code is trivially generalizable to any list of choices. The user selects a row by clicking on a check box produced by a toggle cell renderer.

To get the installed packages that can be upgraded, we use some of the functions provided by the `utils` package.

```
d <- old.packages()[,c("Package", "Installed", "ReposVer")]
d <- as.data.frame(d)
```

This function will be called on the selected rows. Here, we simply call `install.packages` to update the selected packages.

```
doUpdate <- function(d)  install.packages(d$Package)
```

To display the data frame, we first append a column to the data frame to store the selection information and then create a corresponding `RGtkDataFrame`.

```
n <- ncol(d)
nms <- colnames(d)
d$.toggle <- rep(FALSE, nrow(d))
store <- rGtkDataFrame(d)
```

Our tree view shows each text column using a simple text cell renderer, except for the first column that contains the check boxes for selection.

```
view <- gtkTreeView()
# add toggle
cr <- gtkCellRendererToggle()
view$insertColumnWithAttributes(0, "", cr, active = n)
cr['activatable'] <- TRUE
gSignalConnect(cr, "toggled", function(cr, path, user.data) {
  view <- user.data
  row <- as.numeric(path) + 1
  model <- view$getModel()
  n <- dim(model)[2]
  model[row, n] <- !model[row, n]
}, data=view)
```

The text columns are added in one go:

```
mapply(view$insertColumnWithAttributes, -1, nms,
       list(gtkCellRendererText()), text = 1:n-1)
```

Finally, we connect the store to the model.

```
view$setModel(store)
```

To allow the user to initiate the action, we create a button and assign a callback. We pass in the view, rather than the model, in case the model would be recreated by the `doUpdate` call. In a real application, once a package is upgraded it would be removed from the display.

```
b <- gtkButton("Update packages")
gSignalConnect(b, "clicked", function(w, data) {
  view <- data
  model <- view$getModel()
  n <- dim(model)[2]
```

```
  vals <- model[model[, n], -n, drop=FALSE]
  doUpdate(vals)
}, data=view)
```

Our basic GUI places the view into a box container that also holds the button to initiate the action.

```
w <- gtkWindow(show=FALSE)
w$setTitle("Installed packages that need upgrading")
w$setSizeRequest(300, 300)
g <- gtkVBox(); w$add(g)
sw <- gtkScrolledWindow()
g$packStart(sw, expand=TRUE, fill=TRUE)
sw$add(view)
sw$setPolicy("automatic", "automatic")
g$packStart(b, expand=FALSE)
w$show()
```

Progress cell renderers To visually communicate progress within a cell, both progress bars and spinner animations are supported. These modes correspond to `GtkCellRendererProgress` and `GtkCellRendererSpinner`, respectively.

In the case of `GtkCellRendererProgress`, its `value` property takes a value between 0 and 100 indicating the amount finished, with a default value of 0. Values out of this range will be signaled by an error message. For example,

```
cr <- gtkCellRendererProgress()
cr["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 non-trivial 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, one 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 `GtkTreeView`, it is possible to specify a callback that overrides this step. The callback, of type `GtkTreeCellDataFunc`, is passed arguments for the tree view 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(viewCol, cellRend, model, iter, data) {
  curVal <- model$getValue(iter, 0)$value
  fVal <- sprintf("%.3f", curVal)
  cellRend['text'] <- fVal
  cellRend['xalign'] <- 1
}
```

The function then needs to be registered with a `GtkTreeViewColumn` that is rendering a numeric column from the model:

```
view <- gtkTreeView(rGtkDataFrame(data.frame(rnorm(100))))
cr <- gtkCellRendererText()
view$insertColumnWithAttributes(0, "numbers", cr, text = 0)
vc <- view$getColumn(0)
vc$setCellDataFunc(cr, func)
```

The last line is the key: calling `setCellDataFunc` registers our custom formatting function with the view column.

One drawback with the use of such functions is that R code is executed every time a cell is rendered. If performance matters, consider pre-converting the data in the model or tweaking the `options` in R for printing real numbers, namely `scipen` and `digits`.

For customizing rendering further, and in the general case beyond `GtkTreeView`, one could implement a new type of `GtkCellRenderer`. See Section **??** for more details on extending GTK+ classes.

9.2 Display of hierarchical data

Although the `RGtkDataFrame` model is a convenient implementation of `GtkTreeModel`, it has its limitations. Primary among them is its lack of support for hierarchical data. GTK+ implements `GtkTreeModel` with `GtkListStore` and `GtkTreeStore`, which respectively store non-hierarchical and hierarchical tabular data. `GtkListStore` is a flat table, while `GtkTreeStore` organizes the table into a hierarchy. Here, we discuss `GtkTreeStore`.

Loading hierarchical data

A tree store is constructed using `gtkTreeStore`. The column types are specified through a character vector at the time of construction. The specification uses "GTypes" such as `gchararray` for character data, `gboolean` for logical data, `gint` for integer data, `gdouble` for numeric data, and `GObject` for GTK+ objects, such as pixbufs.

Example 9.6: Defining a tree
Below, we create a tree based on the `Cars93` dataset, where the car models are organized by manufacturer, i.e., each model row is the child of its manufacturer row:

```
tstore <- gtkTreeStore("gchararray")
by(Cars93, Cars93$Manufacturer, function(df) {
  piter <- tstore$append()              # parent
  tstore$setValue(piter$iter, column = 0, value =
                  df$Manufacturer[1])
  sapply(df$Model, function(model) {
    sibiter <- tstore$append(parent = piter$iter) # child
    if (is.null(sibiter$retval))
      tstore$setValue(sibiter$iter, column = 0, value = model)
  })
})
```

To retrieve a value from the tree store using its path we have:

```
iter <- tstore$getIterFromString("0:0")
tstore$getValue(iter$iter, column = 0)$value
```

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

This obtains the first model from the first manufacturer.

As shown in this example, populating a tree store relies on two functions: `append`, for appending rows, and `setValue`, for setting row values. The iterator to the parent row is passed to `append`. A parent of `NULL`, the default, indicates that the row should be at the top level. It would also be possible to insert rows using `insert`, `insertBefore`, or `insertAfter`. The `setValue` method expects the row iterator and the column index, 0-based.

An entire row can be assigned through the `set` method. The method uses positional arguments to specify the column and the value. The column index appears as an even argument (say $2k$) and the corresponding value in the odd argument (say $2k + 1$). Values are returned by the `getValue` method, in a list with component `value` storing the value.

Traversing a tree store is most easily achieved through the use of `GtkTreeIter`, introduced previously in the context of flat tables. Here we perform a depth-first traversal of our `Cars93` model to obtain the model values:

```
iter <- tstore$getIterFirst()
models <- NULL
while(iter$retval) {
  child <- tstore$iterChildren(iter$iter)
  while(child$retval) {
    models <- c(models, tstore$get(child$iter, 0)[[1]])
    child$retval <- tstore$iterNext(child$iter)
  }
  iter$retval <- tstore$iterNext(iter$iter)
}
```

The hierarchical structure introduces the method `iterChildren` for obtaining an iterator to the first child of a row. As with other methods returning iterators, the return value is a list, with the `retval` component indicating the validity of the iterator, stored in the `iter` component. The method `iterParent` performs the reverse, iterating from child to parent.

Rows within a store can be rearranged using several methods. Call `swap` to swap rows referenced by their iterators. The methods `moveAfter` and `moveBefore` move one row after or before another, respectively. The `reorder` method totally reorders the rows under a specified parent given a vector of row indices, like that returned by `order`.

Once added, rows may be removed using the `remove` method. To remove every row, call the `clear` method.

Displaying data as a tree

Once a hierarchical dataset has been loaded into a `GtkTreeModel` implementation like `GtkTreeStore`, it can be passed to a `GtkTreeView` widget for display as a tree. Indeed, this is the same widget that displayed our flat data frame in the previous section. As before, `GtkTreeView` displays the `GtkTreeModel` as a table; however, it now adds controls for expanding and collapsing nodes where rows are nested.

The user can click to expand or collapse a part of the tree. These actions correspond to the signals `row-expanded` and `row-collapsed`, respectively.

Example 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.

Now, we make a simple rectangular store for the model information with the following:

```
store <- rGtkDataFrame(Cars93[,"Model", drop=FALSE])
```

Creating a basic view is similar to that for rectangular data already presented:

```
view <- gtkTreeView()
view$insertColumnWithAttributes(0, "Make",
           gtkCellRendererText(), text = 0)
```

```
[1] 1
```

Finally, we illustrate that the same view can be used with either model:

```
view$setModel(store)                # the rectangular store
view$setModel(tstore)               # or the tree store
```

Example 9.8: Dynamically growing a tree
This example uses a tree to explore an R list object, such as that returned by one of R's modeling functions. As the depth of these lists is not specified in advance, we use a dynamic approach to that populates the tree store when a node is expanded and removes nodes when their parent is collapsed.

We begin by defining our tree store and an accompanying tree view. This example allows sorting, and so creates a sort proxy model:

```
store <- gtkTreeStore(rep("gchararray", 2))
sstore <- gtkTreeModelSort(store)
```

We create a root row:

```
iter <- store$append(parent=NULL)$iter
store$setValue(iter, column=0, "GlobalEnv")
store$setValue(iter, column=1, "environment")
iter <- store$append(parent=iter)
```

It is necessary to append an empty row to the root so that root becomes expandable.

We now define the tree view and allow for multiple selection:

```
view <- gtkTreeView(sstore)
view$getSelection()$setMode("multiple")
```

The basic idea is to create child nodes when the parent is expanded and to delete the children when the parent is collapsed. This relies on the `row-expanded` and `row-collapsed` signals, respectively. First, we define the expansion handler:

```
gSignalConnect(view, signal = "row-expanded",
               f = function(view, iter, tpath, user.data) {
                 sortedModel <- view$getModel()
                 iter <- pathToIter(sortedModel, tpath)
```

```
                path <- iterToRPath(sortedModel, iter)
                children <- getChildren(path)
                addChildren(store, children, parentIter=iter)
                ## remove errant offspring, cf. addChildren
                ci <- store$iterChildren(iter)
                if(ci$retval) store$remove(ci$iter)
              })
```

The callback calls several helper functions to map the tree path to an R object, get the child components of the object and add them to the tree. The details are in the definitions of the helper functions.

The `pathToIter` function finds the iterator in the base tree model for a tree path in the sorted proxy.

```
pathToIter <- function(sstore, tpath) {
  store <- sstore$getModel()
  uspath <- sstore$convertPathToChildPath(tpath)
  store$getIter(uspath)$iter
}
```

We now need to convert the iterator to an "R path," which is made up of the names of each component in the list. This function returns such a path given an iterator:

```
iterToRPath <- function(sstore, iter) {
  store <- sstore$getModel()
  indices <- store$getPath(iter)$getIndices()
  iter <- NULL
  path <- sapply(indices, function(i) {
    iter <<- store$iterNthChild(iter, i)$iter
    store$getValue(iter, 0)$value
  })
  return(path[-1])
}
```

The `getChildren` function obtains the child components of a given R object path. If the path is empty, the children are the objects in the global environment, the root. The return value is a `data.frame` with three columns: object name, object class and whether the object is recursive.

```
getChildren <- function(path=character(0)) {
  hasChildren <- function(obj)
    (is.list(obj) || is.environment(obj)) &&
  !is.null(names(as.list(obj)))

  getType <- function(obj) head(class(obj), n=1)

  obj <-
    if (!length(path)) {
      .GlobalEnv
```

```
    } else {
      x <- get(path[1], envir=.GlobalEnv)
      if(length(path) > 1)
        get(path[1], envir=.GlobalEnv)[[path[-1]]]
      else
        x
    }

  children <- as.list(obj)

  d <- data.frame(children = names(children),
                  class = sapply(children, getType),
                  offspring = sapply(children, hasChildren))

  ## filter out Gtk ones
  d[!grepl("^Gtk", d$class), ]
}
```

The final step in the expansion handler is to add the children to the tree store with the `addChildren` function. Its one quirk is the addition of a dummy child row when the item has children. This makes the node expandable, i.e., the tree view draws an icon for the user to click to request the expansion.

```
addChildren <- function(store, children, parentIter = NULL) {
  if(nrow(children) == 0)
    return(NULL)
  for(i in 1:nrow(children)) {
    iter <- store$append(parent=parentIter)$iter
    sapply(1:(ncol(children) - 1), function(j)
           store$setValue(iter, column = j-1, children[i, j]))
    ## Add a branch if there are children
    if(children[i, "offspring"])
      store$append(parent=iter)
  }
}
```

Next, we define a handler for the `row-collapsed` signal, which has a similar signature as the `row-expanded` signal. The handler removes the children of the newly collapsed node, so that we can add them again when the node is expanded.

```
gSignalConnect(view, signal = "row-collapsed",
       f = function(view, iter, tpath, user.data) {
         sortedModel <- view$getModel()
         iter <- pathToIter(sortedModel, tpath)
         n = store$iterNChildren(iter)
         if(n > 1) { ## n=1 gets removed when expanded
           for(i in 1:(n-1)) {
```

```
            child.iter <- store$iterChildren(iter)
            if(child.iter$retval)
              store$remove(child.iter$iter)
          }
        }
      })
```

Our last handler simply demonstrates the retrieval of an object when its row is activated, i.e., double-clicked:

```
gSignalConnect(view, signal = "row-activated",
       f = function(view, tpath, tcol) {
         sortedModel <- view$getModel()
         iter <- pathToIter(sortedModel, tpath)
         path <- iterToRPath(sortedModel, iter)
         sel <- view$getSelection()
         out <- sel$getSelectedRows()
         if(length(out) == 0) return(c()) # nothing
         vals <- c()
         for(i in out$retval) {  # multiple selections
           iter <- sortedModel$getIter(i)$iter
           newValue <- sortedModel$getValue(iter, 0)$value
           vals <- c(vals, newValue)
         }
         print(vals)  # [Replace this]
       })
```

To finish this example, we would need to populate the tree view with columns and display the view in a window.

9.3 Model-based combo boxes

Basic combo box usage was discussed in Section **??**; here we discuss the more flexible and complex approach of using an explicit data model for storing the menu items. The item data is tabular, although it is limited to a single column. Thus, `GtkTreeModel` is again the appropriate model, and `RGtkDataFrame` is usually the implementation of choice.

To construct a `GtkComboBox` based on a user-created model, one should pass the model to the constructor `gtkComboBox`. This model may be changed or set through the `setModel` method and is returned by `getModel`. It remains to instruct the combo box how to display one or more data columns in the menu. Like `GtkTreeViewColumn`, `GtkComboBox` implements the `GtkCellLayout` interface and thus delegates the rendering of model values to `GtkCellRenderer` instances that are packed into the combo box.

The `getActiveIter` returns a list containing the iterator pointing to the currently selected row in the model. If no row has been selected,

the retval component of the list is FALSE. The setActiveIter sets the currently selected item by iterator. As discussed previously, the getActive and setActive behave analogously with 0-based indices.

As introduced in the previous chapter, the GtkComboBoxEntry widget extends GtkComboBox to provide an entry widget for the user to enter arbitrary values. To construct a combo box entry on top of a tree model, one should pass the model, as well as the column index that holds the textual item labels, to the gtkComboBoxEntry constructor. It is not necessary to create a cell renderer for displaying the text, as the entry depends on having text labels and thus enforces their display. It is still possible, of course, to add cell renderers for other model columns.

When a user selects a value with the mouse, the changed signal is emitted. For combo box entry widgets, the changed signal will also be emitted when a new value has been entered. To detect when the user has finished entering text, one needs to retrieve the underlying GtkEntry widget with getChild and connect to its activate signal.

Example 9.9: A combo box with memory

This example uses an editable combo box as an 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. We create the combo box with this model using the first column for the text:

```
m <- rGtkDataFrame(data.frame(
              fname=character(0), visits=character(0),
              novisits=integer(0), stringsAsFactors=FALSE))
cb <- gtkComboBoxEntryNewWithModel(m, 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 tree view. 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:

```
cr <- gtkCellRendererText()
cb$packStart(cr)
cb$addAttribute(cr, "text", 1)
cr['foreground'] <- "gray50"
cr['ellipsize'] <- "end"
cr['style'] <- "italic"
cr['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 round, then calls `help`.

```
callHelpFunction <- function(cb, value) {
  model <- cb$getModel()
  ind <- match(value,model[,1, drop=TRUE])
  n <- model[ind, "novisits"] <- model[ind, "novisits"] + 1
  model[ind, "visits"] <-
    sprintf(ngettext(n, "%s visit", "%s visits"), n)
  ## select for easier editing
  cb$getChild()$selectRegion(start=0,end=-1)
  help(value)
}
gSignalConnect(cb, "changed", f=function(w, ...) {
  if(cb$getActive() >= 0) {
    val <- w$getActiveText()
    callHelpFunction(w, val)
  }
})
```

When the user enters a new value in the entry, we need to check if we have previously accessed the item. If not, we add the value to our model.

```
gSignalConnect(cb$getChild(), "activate",
               f = function(cb, entry, ...) {
  val <- entry$getText()
  if(!any(val == cb$getModel()[,1, drop=TRUE])) {
    model <- cb$getModel()
    model$appendRows(data.frame(fname=val, visits="", novisits=0,
                                stringsAsFactors = FALSE))
  }
  callHelpFunction(cb, val)
}, data=cb, user.data.first=TRUE)
```

We place this in a minimal GUI with a label:

```
w <- gtkWindow(show=FALSE)
w['border-width'] <- 15
g <- gtkHBox(); w$add(g)
g$packStart(gtkLabel("Help on:"))
g$packStart(cb, expand=TRUE, fill=TRUE)
#
w$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 add the common prefix automatically to the entry widget; `popup-single-match` is a logical indicating if a popup is displayed on a single match; `minimum-key-length` takes an integer specifying the number of characters needed in the entry before completion is checked (the default is 1).

By default, the rows in the data model that match the current value of the entry widget in a case insensitive manner are displayed. This matching function can be overridden by setting a new R function through the `setMatchFunc` method. The signature of this function is the completion object, the string from the entry widget (lower case), an iterator pointing to a row in the model and optionally user data that is passed through the `func.data` argument of the `setMatchFunc` method. This callback should return `TRUE` or `FALSE` depending on whether that row should be displayed in the set of completions.

Example 9.10: 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:

```
store <- rGtkDataFrame(state.name)
completion$setModel(store)
completion$setTextColumn(0)
completion['inline-completion'] <- TRUE
completion['popup-single-match'] <- FALSE
```

We wish for the text search to match against any part of a string, not only the beginning, so we define our own match function:

```
matchAnywhere <- function(comp, str, iter, user.data) {
  model <- comp$getModel()
  rowVal <- model$getValue(iter, 0)$value # column 0 in model

  str <- comp$getEntry()$getText()      # case sensitive
  grepl(str, rowVal)
}
completion$setMatchFunc(matchAnywhere)
```

We get the string from the entry widget, not the passed value, as the latter has been standardized to lower case.

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()
```

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 may 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 9.11: Basic gtkTextView usage
The steps to construct a text view consist of:

```
w <- gtkWindow()
w['border-width'] <- 15
#
tv <- gtkTextView()
sw <- gtkScrolledWindow()
sw$setPolicy("automatic", "automatic")
#
w$add(sw)
```

To set all the text in the buffer requires accessing the underlying buffer:

```
buffer <- tv$getBuffer()
buffer$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 may be done as follows:

```
start <- buffer$getStartIter()$iter
end <- buffer$getEndIter()$iter
buffer$getText(start, end)
```

```
[1] "Lorem ipsum dolor sit amet ..."
```

Text may 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.

By default, the text in a view is editable. This can be disabled through the editable property. Typically, one then sets the cursor-visible property to "FALSE" so that the cursor is hidden:

```
tv['editable'] <- FALSE
tv['cursor-visible'] <- FALSE
```

Formatting The text view supports several general formatting options. Automatic line wrapping is enabled through setWrapMode, which takes values from GtkWrapMode: "none", "char", "word", or "word_char". The justification for the entire buffer is controlled by the justification property which takes values of "left", "right", "center", or "fill" from GtkJustification. The global value may be overridden for parts of the

text buffer through the use of text tags, 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 globally set 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 3 parts: the first is a comma-separated list of font families, the second a white-space separated list of style options, and the third a size in points or pixels if the units "px" are included. A typical value might look like `"serif, monospace bold italic condensed 16"`. The various style options are enumerated in `PangoStyle`, `PangoVariant`, `PangoWeight`, `PangoStretch`, and `PangoGravity`. The help page for `PangoFontDescription` contains more information.

9.7 Text buffers

Text buffer properties include `text` for the stored text and `has-selection` to indicate if text is currently selected in a view. The buffer also tracks if it has been modified. This information is available through the buffer `getModified` method, which returns `TRUE` if the buffer has changed. To clear this state, such as when a buffer has been saved to disk, one can pass `FALSE` to `setModified`.

In order to do more with a text buffer, such as retrieve a selection, or modify text attributes, one needs to become familiar with the two mechanism 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

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. In GTK+ a *text iterator* is the primary means of specifying a position in a buffer.

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: 0x103b28d90>
attr(,"interfaces")
character(0)
attr(,"class")
[1] "GtkTextIter" "GBoxed"      "RGtkObject"

$end
<pointer: 0x1057ac330>
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 the component `retval` will be `FALSE`, otherwise it is `TRUE`.

One 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 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()
```

```
[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 which 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()
```

```
[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. One can move an iterator forward or backward, stopping a certain type of landmark. Supported landmarks include characters (`forwardChar`, `forwardChars`, `backwardChar`, and `backwordChars`), words (`forwardWordEnd`, `backwardWordStart`), and sentences (`backwardSentenceStart` and `forwardSentenceEnd`). There are also various methods, such as `insideWord`, for determining the textual context of the iterator. Example 9.12 shows how some of the above are used, in particular how these methods update the iterator rather than return a new one.

Example 9.12: Finding the word that is clicked by the user
This example shows how one can find the iterator corresponding to a mouse click. We obtain the X and Y coordinates of the mouse button press event, find the corresponding iterator, and retrieve the surrounding word:

```
ID <- gSignalConnect(tv, "button-press-event",
                     f=function(w, e, ...) {
  siter <- w$getIterAtLocation(e$getX(), e$getY())$iter
  niter <- siter$copy()                  # need copy
  siter$backwardWordStart()
  niter$forwardWordEnd()
  val <- siter$getText(niter)
  print(val)                             # replace
  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. One can think of a mark as an invisible object stuck between two characters. A mark corresponds to a specific position, like an iterator, except its gravity sets it either to the left or right of the character. 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 to find the corresponding iterator:

```
insertIter <- buffer$getIterAtMark(insert)$iter
bounds$start$getText(insertIter)
```

```
[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 one relates it to the behavior of the text cursor, which has right gravity:

```
insertIter$getOffset()
```

```
[1] 36
```

```
buffer$insert(insertIter, "insertion")
buffer$getIterAtMark(insert)$iter$getOffset()
```

```
[1] 45
```

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)
```

By setting `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.b <- buffer$createTag(tag.name="bold",
                          weight=PangoWeight["bold"])
tag.em <- buffer$createTag(tag.name="em",
                           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.b, iter$start, iter$end)  # updates iters
buffer$applyTagByName("em", iter$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 directly move the marks around, calling `selectRange` is more efficient and convenient. Here, we select the first word:

```
siter <- buffer$getStartIter()$iter
eiter <- siter$copy(); eiter$forwardWordEnd()
```

```
[1] TRUE
```

```
buffer$selectRange(siter, eiter)
```

`GtkTextBuffer` provides some convenience methods for interaction with the clipboard: `copyCliboard`, `cutClipboard` and `pasteClipboard`. To use these, we first need a clipboard object:

```
cb <- gtkClipboardGet()
```

We can then, for example, copy the selected text (the first word) and paste it at the end:

```
buffer$copyClipboard(cb)
buffer$pasteClipboard(cb, override.location = buffer$getEndIter()$iter,
                      default.editable = TRUE)
```

The `default.editable` indicates that the pasted text should be editable. If we had passed `"NULL"`, to the `override.location` argument, the insertion would have occured at the cursor.

Inserting non-text items

If desired, one 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.

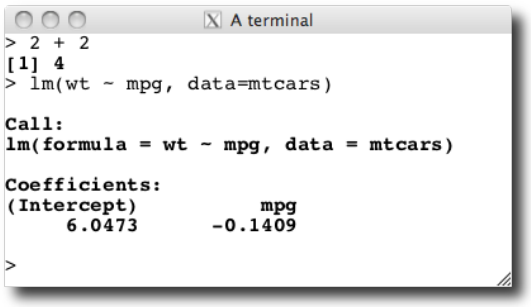

Figure 9.2: A basic R terminal implemented using a `gtkTextView` widget.

Arbitrary child widgets, like a button, can also be inserted. First, one must create an anchor in the text buffer with `createChildAnchor`:

```
anchor <- buffer$createChildAnchor(buffer$getEndIter()$iter)
```

To add the widget, we call the text view method `addChildAtAnchor`:

```
b <- gtkButton("click me")
tv$addChildAtAnchor(b, anchor)
```

Example 9.13: A simple command line interface
This example shows how to create a simple command line interface with the text view widget. 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 text view widget and retrieving its buffer. We also specify a fixed-width font for the buffer.

```
tv <- gtkTextView()
tb <- tv$getBuffer()
font <- pangoFontDescriptionFromString("Monospace")
tv$modifyFont(font)                    # widget wide
```

We will use a few formatting tags, defined next. We do not need the tag objects as variables in the workspace, as we refer to them later by name.

```
tb$createTag(tag.name="cmdInput")
tb$createTag(tag.name="cmdOutput",
             weight=PangoWeight["bold"])
tb$createTag(tag.name="cmdError",
             weight=PangoStyle["italic"], foreground="red")
tb$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:

```
startCmd <- tb$createMark("startCmd", tb$getStartIter()$iter,
                          left.gravity=TRUE)
bufferEnd <- tb$createMark("bufferEnd", tb$getEndIter()$iter)
```

This helper function shows how to move the viewport to the end of the buffer so that the command line is visibile:

```
scrollViewport <- function(view, ...) {
  iter <- view$getBuffer()$getEndIter()$iter
  view$scrollToMark(bufferEnd, within.margin=0)
  return(FALSE)
}
```

There are two types of prompts needed: one for entering a new command and one for a continuation. This function adds either, depending on its argument:

```
addPrompt <- function(obj, prompt=c("prompt", "continue"),
                      setMark=TRUE)
{
  prompt <- match.arg(prompt)
  prompt <- getOption(prompt)

  endIter <- obj$getEndIter()
  obj$insert(endIter$iter, prompt)
  if(setMark)
    obj$moveMarkByName("startCmd", endIter$iter)
  obj$applyTagByName("uneditable", obj$getStartIter()$iter,
                     obj$getEndIter()$iter)
}
addPrompt(tb) ## place an initial prompt
```

This helper method writes the output of a command to the text buffer:

```
addOutput <- function(obj, output, tagName="cmdOutput") {
  endIter <- obj$getEndIter()
  if(length(output) > 0)
    sapply(output, function(i)  {
      obj$insertWithTagsByName(endIter$iter, i, tagName)
      obj$insert(endIter$iter, "\n", len=-1)
    })
}
```

We arrange to truncate large outputs. By passing in the tag name, we can specify whether this is normal output or an error message.

This next function uses the `startCmd` mark and the end of the buffer to extract the current command:

```
findCMD <- function(obj) {
  endIter <- obj$getEndIter()
```

```
  startIter <- obj$getIterAtMark(startCmd)
  cmd <- obj$getText(startIter$iter, endIter$iter, TRUE)
  regex <- paste("\n[", getOption("continue"), "] ", sep = "")
  cmd <- unlist(strsplit(cmd, regex))
  cmd
}
```

The "regex" is used to parse multi-line commands.

The following function takes the current command and evaluates it. It uses a hack (involving grep) to distinguish between an incomplete command and a true syntax error.

```
evalCMD <- function(tv, cmd) {
  tb <- tv$getBuffer()
  cmd <- paste(cmd, sep="\n")
  out <- try(parse(text=cmd), silent=TRUE)
  if(inherits(out, "try-error")) {
    if(length(grep("end", out))) {      # unexpected end
      ## continue
      addPrompt(tb, "continue", setMark=FALSE)
    } else {
      ## error
      addOutput(tb, out, tagName = "cmdError")
    }
    scrollViewport(tv)
    return()
  }

  e <- parse(text = cmd)
  out <- capture.output(vis <-
                withVisible(try(eval(e, .GlobalEnv), TRUE)))

  addOutput(tb, out)
  if (inherits(vis$value, "try-error"))
    addOutput(tb, vis$value, "cmdError")
  else if (vis$visible)
    addOutput(tb, capture.output(print(vis$value)))

  addPrompt(tb, "prompt", setMark=TRUE)
  scrollViewport(tv)
}
```

The evalCMD command is called when the return key is pressed:

```
gSignalConnect(tv, "key-release-event", f=function(w, e) {
  obj <- w$getBuffer()                  # w is textview
  keyval <- e$getKeyval()
  if(keyval == GDK_Return) {
    cmd <- findCMD(obj)                 # poss. character(0)
```

```
    if(length(cmd) && nchar(cmd) > 0)
      evalCMD(w, 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:

Figure 9.2 shows the widget placed into a very simple GUI.

10

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:

```
a <- gtkAction(name="ok", label="_Ok",
               tooltip="An OK button", stock.id="gtk-ok")
```

The constructor takes arguments `name` (to programmatically refer to the action), `label` (the displayed text), `tooltip`, and `stock.id` (identifying a stock icon). The command associated with an action is implemented by a callback connected to the `activate` signal:

```
gSignalConnect(a, "activate", f = function(w, data) {
  print(a$getName())                    # or some useful thing
})
```

An action plays the role of a data model describing a command, while widgets that implement the `GtkActivatable` interface are the views and controllers. All buttons, menu items and tool items implement `GtkActi-`

vatable and thus may serve as action proxies. Actions are connected to widgets through the method setRelatedAction:

```
b <- gtkButton()
b$setRelatedAction(a)
```

Certain aspects of a proxy widget are coordinated through the action. This includes sensitivity and visibility, corresponding to the sensitive and visible properties. By default, aesthetic properties like 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(a, "<control>O")
```

In addition to the properties already introduced, an action may have a shorter label for display in a toolbar (short_label), and hints for when to display its label (is_important) and image (always_show_image).

There is a special type of action that has a toggled state: GtkToggleAction. The active property represents the toggle. A further extension is GtkRadioAction, where the toggled state is shared across a list of radio actions, via the group property. Proxy widgets represent toggle and radio actions with controls resembling check boxes and radio buttons, respectively. Here, we create a toggle action for fullscreen mode:

```
fullScreen <- gtkToggleAction("fullscreen", "Full screen",
                              "Toggle full screen")
gSignalConnect(fullScreen, "toggled", function(action) {
  if(fullScreen['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 "popup" when the user interacts with the element, such as by clicking the right mouse button.

Menu bars and popup menus may be constructed by appending each menu item and submenu separately, as illustrated below. For menus with more than a few items, we recommend the strategies described in Section 10.5.

Menu Bars

We will first demonstrate the menu bar, leaving the popup menu for later. 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 can proxy an action. Usually, a menu bar consists multiple instances of the other type of menu shell: the menu, `GtkMenu`. Here, we create a menu object for our "File" menu:

```
fileMenu <- gtkMenu()
```

As a menu is not itself a menu item, we first must embed the menu into a menu item, which is labeled with the menu title:

```
fileItem <- gtkMenuItemNewWithMnemonic(label="_File")
fileItem$setSubmenu(fileMenu)
```

The underscore in the label indicates the key associated with the mnemonic for use when navigating the menu with a keyboard. Finally, we append the item containing the file menu to the menu bar:

```
menubar$append(fileItem)
```

In addition to `append`, it is also possible to `prepend` and `insert` menu items into a menu shell. As with any container, one can `remove` a child menu item, although the convention is to desensitize an item, through the `sensitive` property, when it is not currently relevant.

Next, we populate our file menu with menu items that perform some command. For example, we may desire an open item:

```
open <- gtkMenuItemNewWithMnemonic("_Open")
```

This item does not have an associated `GtkAction`, so we need to implement its activate signal directly:

```
gSignalConnect(open, "activate", function(item) {
  f <- file.choose()
  file.show(f)
})
```

The item is now ready to be added to the file menu:

```
fileMenu$append(open)
```

It is recommended, however, to create menu items that proxy an action. This will facilitate, for example, adding an equivalent toolbar item later. We demonstrate with a "Save" action:

```
saveAction <-
  gtkAction("save", "Save", "Save object", "gtk-save")
```

Then the appropriate menu item is generated from the action and added to the file menu:

```
save <- saveAction$createMenuItem()
fileMenu$append(save)
```

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:

```
fileMenu$append(gtkSeparatorMenuItem())
```

Toggle menu items, i.e., a label next to a check box, are also supported. A toggle action will create one implicitly:

```
autoSaveAction <- gtkToggleAction("autosave", "Autosave",
                                  "Enable autosave")
autoSave <- autoSaveAction$createMenuItem()
fileMenu$append(autoSave)
```

Finally, we add our menu bar to the top of a window:

```
mainWindow <- gtkWindow()
vbox <- gtkVBox()
mainWindow$add(vbox)
vbox$packStart(menubar, FALSE, FALSE)
```

Popup Menus

Example 10.1: Popup menus
To illustrate popup 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 popup a menu when clicked with the third (right) mouse button:

```
b <- gtkButton("Click me with right mouse button")
w <- gtkWindow(); w$setTitle("Popup menu example")
w$add(b)
```

This menu will be shown by calling `gtkMenuPopup` in response to the `button-press-event` signal on the button:

```
gSignalConnect(b,"button-press-event",
       f = function(w, e, menu) {
         if(e$getButton() == 3 ||
            (e$getButton() == 1 && # a mac
             e$getState() == GdkModifierType['control-mask']))
           gtkMenuPopup(menu,
                        button = e$getButton(),
                        activate.time = e$getTime())
         return(FALSE)
       }, data=popup)
```

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 popup 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:

```
IDs <- sapply(popup$getChildren(), function(i) {
  if(!inherits(i, "GtkSeparatorMenuItem")) # skip these
    gSignalConnect(i, "activate",
                   f = function(w, data) print("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 life-time of the application, not upon user click. 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:

```
openButton <- 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 may be added to the toolbar with the `add` method:

```
toolbar$add(openButton)
```

This appends the open button to the end of the toolbar. To insert into a specific position, we would call the `insert` method.

Usually, any application with a toolbar also has a menu bar, in which case many actions are shared between the two containers. Thus, it is often beneficial to construct a tool button directly from its corresponding action:

```
saveButton <- saveAction$createToolItem()
toolbar$add(saveButton)
```

A tool button is created for `saveAction`, created in the previous section.

Like menus, related buttons may be grouped using separators:

```
toolbar$add(gtkSeparatorToolItem())
```

Any toggle action will create a toggle tool button as its proxy:

```
fullScreenButton <- fullScreen$createToolItem()
toolbar$add(fullScreenButton)
```

A `GtkToggleToolButton` embeds a `GtkToggleButton`, which is depressed whenever its `active` property is `TRUE`.

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 only display 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:

```
fullScreen["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:

```
helpAction <- gtkAction("help", "Help", "Get help", "gtk-help")
toolbar$add(helpAction$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.

```
gdkColor <- gdkColorParse(palette()[1])$color
colorButton <- gtkColorButton(gdkColor)
```

`gtkColorButton` requires the initial color to be specified as a `GdkColor`, which we parse from the R color name.

The next step is to build the menu. Each menu item will display a $20x20$ rectangle, filled with the color, next to the color name:

```
colorMenuItem <- function(color) {
  da <- gtkDrawingArea()
  da$setSizeRequest(20, 20)
  da$modifyBg("normal", color)
  item <- gtkImageMenuItem(color)
  item$setImage(da)
  item
}
colorItems <- sapply(palette(), colorMenuItem)
colorMenu <- gtkMenu()
```

```
for (item in colorItems)
  colorMenu$append(item)
```

An important realization is that the image in a GtkImageMenuItem may be any widget that presumably draws an icon; it need not be an actual GtkImage. In this case, we use a drawing area with its background set to the color. When an item is selected, its color will be set on the color button:

```
colorMenuItemActivated <- function(item) {
  color <- gdkColorParse(item$getLabel())$color
  colorButton$setColor(color)
}
sapply(colorItems, gSignalConnect, "activate",
       colorMenuItemActivated)
```

Finally, we place the color button and menu together in the menu tool button:

```
menuButton <- gtkMenuToolButton(colorButton, "Color")
menuButton$setMenu(colorMenu)
toolbar$add(menuButton)
```

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 as one solution, which leaves the configuration of a multi-row toolbar to the user. 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 a two groups of tool items:

```
fileGroup <- gtkToolItemGroup("File")
fileGroup$add(gtkToolButton(stock.id = "gtk-open"))
fileGroup$add(saveAction$createToolItem())
helpGroup <- gtkToolItemGroup("Help")
helpGroup$add(helpAction$createToolItem())
```

The groups are then added to an instance of GtkToolPalette:

```
palette <- gtkToolPalette()
palette$add(fileGroup)
palette$add(helpGroup)
```

Finally, we can programmatically collapse a group:

```
helpGroup$setCollapsed(TRUE)
```

10.4 Status Reporting

Status bars

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

Information bars

An information bar is similar in purpose to a message dialog, only it is intended to be less obtrusive. Typically, an information 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: one places widgets into a content area, and listens to the `response` signal.

We create our info bar:

```
ib <- gtkInfoBar(show=FALSE)
ib$setNoShowAll(TRUE)
```

We call `setNoShowAll`o prevent the widget from being shown when `showAll`s called on the parent. Normally, an information 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`:

```
l <- gtkLabel("Warning, Warning ....")
ib$setMessageType("warning")
ib$getContentArea()$add(l)
```

A button to allow the user to hide the bar can be added as follows:

```
ib$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 handle simply closes the bar:

```
gSignalConnect(ib, "response", function(w, resp.id) w$hide())
```

Finally, we add the info bar to our main window and show it:

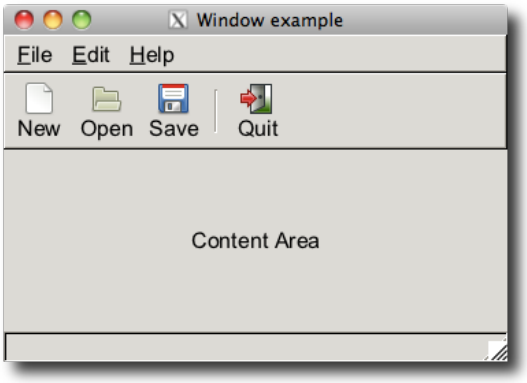

Figure 10.1: A GUI made using a UI manager to layout the menubar and toolbar.

```
vbox$packStart(ib, expand = FALSE)
ib$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.

The steps required to use GTK+'s UI manager are

1. construct the UI manager,
2. define the actions in groups,
3. specify the layout of the menu bars and toolbars,
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

We begin by constructing the UI manager:

```
uimanager = gtkUIManager()
```

Next, we define the action groups. For demonstration purposes, our actions simply push the action name onto the status bar at the bottom of the window:

```
someAction <- function(action,...)
  statusbar$push(statusbar$getContextId("message"),
                 action$getName())
```

We also need a quit handler:

```
Quit <- function(...) win$destroy()
```

We break up our action group definitions into one for "File", and "Edit" and one for "Help." Every action is defined by a `GtkActionEntry` structure, which is represented by a list in R. The components (in order) are the name; the icon; the label, with _ specifying the mnemonic; the keyboard accelerator, with `<control>`, `<alt>`, `<shift>` as possible prefixes, a tooltip, and finally the callback. Empty values can be defined as `NULL` or, except for the callback, an empty string.

We define the actions for the "File" and "Edit" menus:

```
firstActionGroup <- gtkActionGroup("firstActionGroup")
firstActionEntries <- list(
  ## name,ID,label,accelerator,tooltip,callback
  file = list("File",NULL,"_File",NULL,NULL,NULL),
  new = list("New", "gtk-new", "_New", "<control>N",
    "New document", someAction),
  sub = list("Submenu", NULL, "S_ub", NULL, NULL, NULL),
  open = list("Open", "gtk-open", "_Open", "<ctrl>O",
    "Open document", someAction),
  save = list("Save", "gtk-save", "_Save", "<alt>S",
    "Save document", someAction),
  quit = list("Quit", "gtk-quit", "_Quit", "<ctrl>Q",
    "Quit", Quit),
  edit = list("Edit", NULL, "_Edit", NULL, NULL, NULL),
  undo = list("Undo", "gtk-undo", "_Undo", "<ctrl>Z",
    "Undo change", someAction),
  redo = list("Redo", "gtk-redo", "_Redo", "<ctrl>U",
    "Redo change", someAction)
)
```

In the above, we create dummy actions named "File" and "Edit" that perform no function. They are necessary for specifying the menu layout later. We now add the actions to the action group, then add this action group to the first spot in the UI manager:

```
firstActionGroup$addActions(firstActionEntries)
uimanager$insertActionGroup(firstActionGroup, 0) # 0-based
```

The redo feature should only be sensitive to mouse events after a user has undone an action. If we wanted to alter the sensitivity of the redo action, we would need to retrieve it from the action group:

```
redo <- firstActionGroup$getAction("Redo")
redo['sensitive'] <- FALSE
```

It is also possible to define toggle actions, as we demonstrate presently for the "Help" group. First, we define the ordinary actions:

```
helpActionGroup <- gtkActionGroup("helpActionGroup")
helpActionEntries <- list(
  help = list("Help", "", "_Help", "", "", NULL),
  about = list("About", "gtk-about", "_About", "", "",
    someAction)
)
helpActionGroup$addActions(helpActionEntries)
```

Next, we define a "Use tooltips" toggle action:

```
toggleActions <- list(
  tooltips = list("UseTooltips", NULL, "Use _Tooltips", "<control>T",
    "Enable tooltips", someAction, TRUE)
)
helpActionGroup$addToggleActions(toggleActions)
```

```
NULL
```

The list structure for toggle action entry is identical to that of the ordinary actions, except for the last element which indicates whether the action is initially active. One can also incorporate radio actions, although this is not shown. Finally, we insert the help action group in the second position:

```
uimanager$insertActionGroup(helpActionGroup, 1)
```

Our menubar and toolbar layout is specified as 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>
  <menubar name="menubar">
    <menu name="FileMenu" action="File">
      <menuitem name="FileNew" action="New"/>
      <menu action="Submenu">
<menuitem name="FileOpen" action="Open" />
      </menu>
      <menuitem name="FileSave" action="Save"/>
      <separator />
      <menuitem name="FileQuit" action="Quit"/>
    </menu>
    <menu action="Edit">
      <menuitem name="EditUndo" action="Undo" />
      <menuitem name="EditRedo" action="Redo" />
    </menu>
    <menu action="Help">
```

```
    <menuitem action="UseTooltips"/>
    <menuitem action="About"/>
   </menu>
  </menubar>
  <toolbar name="toolbar">
   <toolitem action="New"/>
   <toolitem action="Open"/>
   <toolitem action="Save"/>
   <separator />
   <toolitem action="Quit"/>
  </toolbar>
</ui>
```

The top-level element is named `ui`, only one of which is allowed in a UI definition. The children of `ui` represent a top-level action container: `menubar`, `toolbar`, or `popup`. The `name` attributes are used to refer to the widgets later. The `menubar` element contains `menu` elements, which in turn contain `menuitem` and `separator` elements, as well as additional `menu` elements for nesting. The toolbars are populated with `toolitem` elements. The item elements have an `action` attribute that refers to an action in one of our action groups and an optional name (defaulting to the `action` value).

This file is loaded into the UI manager as follows

```
id <- uimanager$addUiFromFile("ex-menus.xml")
```

The `id` value can be used to merge and delete UI components according to the mode of the UI, but this is not illustrated here.

Now we can setup a basic window template with a menubar, toolbar, and status bar. We first construct the three main widgets. The UI manager will construct our toolbar and menubar, as identified from the names specified in the UI definition:

```
menubar <- uimanager$getWidget("/menubar")
toolbar <- uimanager$getWidget("/toolbar")
```

The statusbar is constructed with

```
statusbar <- gtkStatusbar()
```

Now we create a top-level window and attach a keyboard accelerator group to the window so that when the window has the focus, the keyboard shortcuts defined for our actions are active:

```
win <- gtkWindow(show=TRUE)
win$setTitle("Window example")
accelgroup <- uimanager$getAccelGroup()
win$addAccelGroup(accelgroup)
```

Now it is a simple matter of packing the widgets into a box.

```
box <- gtkVBox()
win$add(box)
box$packStart(menubar, expand=FALSE, fill=FALSE, 0)
box$packStart(toolbar, expand=FALSE, fill= FALSE, 0)
contentArea <- gtkVBox()
box$packStart(contentArea, expand=TRUE, fill=TRUE, 0)
contentArea$packStart(gtkLabel("Content Area"))
box$packStart(statusbar, expand=FALSE, fill=FALSE, 0)
```

Part III

The `qtbase` package

11

Qt: Overview

11.1 The Qt library

Qt is an open-sourced, cross-platform application framework that is perhaps best known for its widget toolkit. The features of Qt are divided into about a dozen modules. We highlight some of the more important and interesting ones:
Core Basic utilties, collections, threads, I/O, ...
Gui Widgets, models, etc for graphical user interfaces
OpenGL Convenience layer (e.g., 2D drawing API) over OpenGL
Webkit Embeddable HTML renderer (shared with Safari, Chrome)
Other modules include functionality for networking, XML, SQL databases, SVG, and multimedia. However, R packages already provide many of those features.

The history of Qt begins with Haavard Nord and Eirik Chambe-Eng in 1991 and follows with the Trolltech company, until 2008. It is now owned by Nokia, a major cell-phone producer. 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 include the communication application Skype and the KDE desktop for Linux.

Qt is developed in C++ with extensions that require a special preprocessor called the *Meta Object Compiler* (`MOC`). The `MOC` allows for convenient syntax in the definition of signals, slots (signal handlers), and properties, which behave very similarly to those of GTK+.

There are many languages with bindings to Qt, and R is one such language. The `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, 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 development of `qtbase` package is hosted on Github (`http://github.com/ggobi/qtbase`). It depends on the Qt framework, available as a binary install from `http://qt.nokia.com/`.

11.2 An introductory example

As a synopsis for how one programs 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.

After ensuring that the underlying libraries are installed, the package may be loaded like any other R package:

```
require(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.

```
window <- Qt$QWidget()
label <- Qt$QLabel("Date:")
edit <- Qt$QLineEdit()
button <- Qt$QPushButton("Ok")
```

The constructors are not found in the global environment, but rather in the `Qt` environment, an object exported from the `qtbase` namespace. As such, the `$` lookup operator is used.

[1] There is a GTK+ WebKit port, but it is not included with GTK+ itself.

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 layout managers to organize widgets. This is similar to Java/Swing and `tcltk`, but not `RGtk2`. Layout managers will be discussed more thoroughly in Chapter 12, but in this example we will use a grid layout to organize our widgets. The placement of child widgets into the grid is done through the `addWidget` method and requires a specification, by index and span, of the cells the child will occupy:

```
lyt <- Qt$QGridLayout()
lyt$addWidget(label, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(edit, 0, 1, 1, 1)
lyt$addWidget(button, 1, 1, 1, 1)
```

One can adjust properties of the layout, but we leave that discussion for later.

We need to attach our layout to the widget `w`:

```
window$setLayout(lyt)
```

Finally, to view our GUI (Figure 11.1), we must call its `show` method.

```
window$show()
```

Callbacks As with other GUI toolkits, we add interactivity to our GUI by binding callbacks to certain 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, "pressed", handler)
```

We will discuss callbacks more completely in Section 11.6.

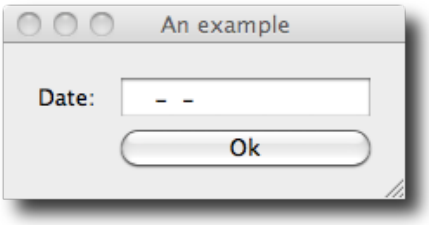

Figure 11.1: Screenshot of our sample GUI to collect a date from the user.

Object-oriented support `QLineEdit` can validate text input, and we would like to validate the entered date. There are a few built-in validators, and 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 `qtbase`. More details on working with classes and methods are provided in Section 11.8. 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$Invalid)
  else
    return(Qt$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 11.7 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)
```

11.3 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 318 public methods
```

An `RQtClass` object contains methods in the class scope (*static* methods in C++), enumerations defined by the class, and additional `RQtClass` objects representing nested classes or namespaces. Here we list some of the components of `QWidget` and access one of the enumeration values:

```
head(names(Qt$QWidget), n = 3)
```

```
[1] "connect"              "DrawChildren"         "DrawWindowBackground"
```

```
Qt$QWidget$DrawChildren
```

```
Enum value: DrawChildren (2)
```

Most importantly, however, an instance of `RQtClass` is in fact an R `function` object, and serves as the constructor of instances of the class. For example, we could construct an instance of `QWidget`:

```
w <- Qt$QWidget()
```

The `w` object has a class structure that reflects the class inheritance structure of Qt:

```
class(w)
```

```
[1] "QWidget"             "QObject"             "QPaintDevice"
[4] "UserDefinedDatabase" "environment"         "RQtObject"
```

[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.

The base class, `RQtObject`, is an environment containing the properties and methods of the instance. For `w`, we list the first few using `ls`:

```
head(ls(w), n=3)
```

```
[1] "heightForWidth" "macQDHandle"    "repaint"
```

Properties and methods are accessed from the environment in the usual manner. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window:

```
w$windowTitle                                  # initially NULL
```

```
NULL
```

```
w$windowTitle <- "a new title"
w$windowTitle
```

```
[1] "a new title"
```

Although Qt defines methods for accessing properties, the R user will normally invoke methods that perform some action. For example, we could show our widget:

```
w$show()
```

The environment structure of the object masks the fact that the properties and methods may be defined in a parent class of the object. For example, a button widget is provided by the `QPushButton` constructor, as in

```
b <- Qt$QPushButton()
```

`QPushButton` extends `QWidget` and thus inherits the properties like `windowTitle`:

```
is(b, "QWidget")
```

```
[1] TRUE
```

```
b$windowTitle
```

```
NULL
```

It is important to realize this distinction when referencing the documentation. As with GTK+, the methods are documented with the class that declares the method.

11.4 Methods and dispatch

In C++, it is possible to have multiple methods and constructors with the same name, but different signatures. This is called *overloading*. An overloaded method is roughly similar to an S4 generic, save the obvious difference that an S4 generic does not belong to any class. The selected overload is that with the signature that best matches the types of the arguments. The exact rules of overload resolution are beyond our scope.

It is particularly common to overload constructors. For example, a simple push button can be constructed in several different ways. Here again is the invocation of the `QPushButton` constructor with no arguments:

```
b <- Qt$QPushButton()
```

By convention, all classes derived from `QObject`, including `QWidget`, provide a constructor that accepts a parent `QObject`. This has important consequences that are discussed later. We demonstrate this for `QPushButton`:

```
w <- Qt$QWidget()
b <- Qt$QPushButton(w)
```

An alternative constructor for `QPushButton` accepts the text for the label on the button:

```
b <- Qt$QPushButton("Button text")
```

Buttons may also have icons, for example

```
icon <- Qt$QIcon(system.file("images/ok.gif", package="gWidgets"))
b <- Qt$QPushButton(icon, "Ok")
```

We have passed three different types of object as the first argument to `Qt$QPushButton`: a `QWidget`, a string, and finally a `QIcon`. The dispatch depends only on the type of argument, unlike the constructors in `RGtk2`, which dispatch based on which arguments are specified. (In particular, dispatch is based on position of argument, but not on names given to arguments. We use names only for clarity in our examples.)

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:

```
out <- qmethods(Qt$QPushButton)
dim(out)
```

```
[1] 435   5
```

```
head(out, n=3)
```

```
         name       return                    signature protected static
1 QPushButton QPushButton*               QPushButton()     FALSE
FALSE
2 QPushButton QPushButton*       QPushButton(QWidget*)     FALSE
FALSE
3 QPushButton QPushButton* QPushButton(QIcon, QString)     FALSE
FALSE
```

11.5 Properties

Every `QObject`, which includes every widget, may declare a set of properties that represent its state. We list some of the available properties for our button:

```
head(qproperties(b))
```

```
                            type readable writable
objectName                 QString     TRUE     TRUE
modal                         bool     TRUE    FALSE
windowModality Qt::WindowModality     TRUE     TRUE
enabled                       bool     TRUE     TRUE
geometry                     QRect     TRUE     TRUE
frameGeometry                QRect     TRUE    FALSE
```

As shown in the table, every property has a type and logical settings for whether the property is readable and/or writeable. Virtually every property value may be read, and it is common for properties to be read-only. For example, we can fully manipulate the `objectName` property, but our attempt to modify the `modal` property fails:

```
b$objectName <- "My button"
b$objectName
```

```
[1] "My button"
```

```
b$modal
```

```
[1] FALSE
```

```
try(b$modal <- TRUE)                    # fails
```

Qt provides accessor methods for getting and setting properties. The getter methods have the same name as the property, so they are masked at the R level. Setter methods are available and are typically named with the word "set" followed by the property name:

```
b$setObjectName("My button")
```

However, it is recommended to use the replacement syntax shown in the previous example, for the sake of symmetry.

11.6 Signals

Qt in C++ uses an architecture of signals and slots to have components communicate with each other. A component emits a signal when some event happens, such as a user clicking on a button. Qt allows one to define a special type of method known as a slot in another component (or the same) that can be connected to the signal as the handler. The two components are decoupled as the emitter does not need to know about the receiver except through the signal connection. In R, any function can be treated as a slot and connected as a signal handler. This is similar to the signal handling in `RGtk2`. The function `qconnect` establishes the connection of an R function to a signal. For example

```
b <- Qt$QPushButton("click me")
qconnect(b, "clicked", function() print("ouch"))
b$show()
```

Signals are defined by a class and are inherited by subclasses. Here, we list some of the available signals for the `QPushButton` class:

```
tail(qsignals(Qt$QPushButton), 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, the following two calls are equivalent:

```
qconnect(b, "clicked", function(checked) print(checked))
qconnect(b, "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.

The `qconnect` function returns a dummy `QObject` instance that provides the slot that wraps the R function. This dummy object can be used with the `disconnect` method on the sender to break the signal connection:

```
proxy <- qconnect(b, "clicked", function() print("ouch"))
b$disconnect(proxy)
```

```
[1] TRUE
```

One can block all signals from being emitted with the `blockSignals` method, which takes a logical value to toggle whether the signals should be blocked.

Unlike GTK+, Qt widgets generally do not emit hardware events, such as a mouse press, via signals. Instead, a method in the widget is invoked upon receipt of an event. The developer is expected to extend the widget and override the method to catch the event. The apparent philosophy of Qt is that hardware events are low-level and thus should be handled by the widget, not some other instance. We will discuss extending classes in Section **??**. Example **??** demonstrates handling widget events.

11.7 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 specifed directly as integers, they are given the class `QtEnum` and have the overloaded operators | and & to combine values bitwise. This makes the most sense when the values correspond to bit flags, as is the case for the alignment style. For example, aligning the text in a label in the upper right can be done through

```
l <- Qt$QLabel("Our text")
l$alignment <- Qt$Qt$AlignRight | Qt$Qt$AlignTop
```

To check if the alignment is to the right, we could query by:

```
as.logical(l$alignment & Qt$Qt$AlignRight)
```

```
[1] FALSE
```

11.8 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 only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbase` package allows the R user to extend C++ classes in order to enhance the features of Qt. 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. Properties 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 a storage mechanism 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 320 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$QWidget,
          function(title, prop, parent=NULL) {
            super(parent)
            this$property <- prop
            setWindowTitle(title)
          })
```

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

One may define new methods, or override methods from a base class through the `qsetMethod` function. For example, accessors for a field may 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"`, `"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)
```

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

If the signal takes an argument, we need to indicate that in the signature:

```
qsetSignal("somethingHappenedAtIndex(int)", SubClass)
```

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

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) {
  # ....
})
```

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

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 one is using the automated connection feature, see Section **??**.

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
```

```
NULL
```

```
instance$property <- "value"
instance$property
```

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

However, the property is not actually exposed by the Qt meta object system. That requires specifying the `type` argument, which we will cover later.

By default, the property value is actually stored as a (private) field in the object, called `".property"`. One can override the default behavior by specifying a function for the getter and/or the setter:

```
qsetProperty("checkedProperty", SubClass, write = function(x) {
  if (!is(x, "character"))
    stop("'checkedProperty' must be a character vector")
  this$.checkedProperty <- x
})
```

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

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. One might also want to override the `read` function, for cases where a property depends only on other properties or on some external resource.

To automatically emit a signal whenever a property is set, one can pass the name of the signal as the `notify` of `qsetProperty`:

```
qsetSignal("propertyChanged", SubClass)
```

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

```
qsetProperty("property", SubClass, notify = "propertyChanged")
```

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

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     TRUE     TRUE
```

We see that the type is now exposed via the general `qproperties` function. Specifying the type enables all of the features of a Qt property.

Example 11.1: A model for workspace objects
In this example, we revisit creating a backend storage model for a workspace browser (cf. Example 4.6). With `gWidgets`, we needed to define an Observable class for our model. With Qt, we can leverage the existing signal framework to notify views of changes to the model. This example demonstrates only the model; implementing a view is left to the reader.

Our basic model subclasses `QObject`, not `QWidget`, as it has no graphical representation – a job left for its views:

```
qsetClass("WSModel", Qt$QObject, function(parent=NULL) {
  super(parent)
  updateVariables()
})
```

```
Class 'R::.GlobalEnv::WSModel' with 50 public methods
```

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 digest in a property:

```
qsetProperty("digest", WSModel)
```

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

When a new object is added, an object is deleted, or an object is changed, we wish to signal that occurence to any views of the model. For that purpose, we define a new signal below:

```
qsetSignal("objectsChanged", WSModel)
```

We then specify this signal name to the `notify` argument when defining the `objects` property, so that assignment will emit the signal:

```
qsetProperty("objects", WSModel, notify="objectsChanged")
```

Our class has a method to update the variable list. This simply compares the digest of the current workspace objects with a cached list. If there are differences, we update the objects, which in turn signals a change.

```
qsetMethod("updateVariables", WSModel, function() {
  x <- sort(ls(envir=.GlobalEnv))
```

```
  objs <- sapply(x, function(i)
               digest(get(i, envir=.GlobalEnv)))

  if((length(objs) != length(digest)) ||
     length(digest) == 0 ||
     any(objs != digest)) {
    this$digest <- objs       # update cache
    this$objects <- x         # emit signal
  }
  invisible()
})
```

To illustrate, we create a notifier that the obejcts were changed by assigning a callback:

```
m <- WSModel()
qconnect(m, "objectsChanged", function()
         message("workspace objects were updated"))
```

Finally, we arrange to call the update function as needed. If the workspace size is modest, using a task callback is a reasonable strategy:

```
addTaskCallback(function(expr, value, ok, visible) {
  m$updateVariables()
  TRUE
})
```

11.9 `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 heirarchy. 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.

```
w <- Qt$QWidget()
w$visible
```

```
[1] FALSE
```

```
w$visible <- TRUE
w$visible
```

```
[1] TRUE
```

The show and hide methods are the corresponding convenience functions for making a widget visible and invisible, respectively.

```
w$show()
```

```
w$visible
```

```
[1] TRUE
```

```
w$hide()
```

```
w$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.

Similary, the property enabled controls whether a widget is sensitive to user input, including mouse events.

```
b <- Qt$QPushButton("button")
b$enabled <- FALSE
b$enabled
```

```
[1] FALSE
```

Only one widget can have the keyboard focus at once. The user shifts the focus by tab-navigation or mouse clicks (unless customized, see focusPolicy). When a widget has the focus, its focus property is TRUE. The property is read-only; the focus may be shifted 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 user hovers the mouse over the widget. Similarly, the statusTip property holds a string to be shown in the statusbar instead of a popup window. Finally, Qt provides a "What's This?" tool that will show the text in the whatsThis property in response to 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 12. To specify the size of a top-level window, manipulate the size property, which holds a QSize object:

```
w$size <- qsize(400, 400)
## or
w$resize(400, 400)
w$show()
```

We create the QSize object with the qsize convenience function implemented by the qtbase package. The resize method is another convenient shortcut. One should generally configure the size of a window before showing it. This helps the window manager optimally place the window.

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:

```
f <- qfont(family="helvetica", pointsize=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:

```
l <- Qt$QLabel("Text for the label")
l$font$toString()
l$font <- f
l$font$toString()
```

The QFont class has several methods to query the font and to adjust properties. For example, there are the methods setFamily, setUnderline, setStrikeout and setBold among others.

Styles

Palette Every platform has its own distinct look and feel, and an application should 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 specific role, as enumerated in QPalette::ColorRole. Examples include the color for 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. One can change the colors used in rendering a widget by manipulating the palette, as illustrated in Example **??**.

Style Sheets Cascading style sheets (CSS) are used by web deisgners 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 stylesheets 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:

```
b <- Qt$QPushButton("Style sheet example")
b$show()
b$styleSheet <- "QPushButton {color: red; background: white}"
```

The CSS syntax may be unfamiliar to R programmers, so the `qtbase` package provides an alternative interface that is reminiscent of the `par` function. We specify the above stylesheet in this syntax:

```
qsetStyleSheet(color = "red", background = "white",
               what = "QPushButton", widget = b)
```

The `widget` argument defaults to `NULL`, which applies the stylesheet application-wide through the `QApplication` instance. The default for `what` is `"*"`, meaning that the stylesheet applies to any widget class. The following would cause all widgets in the application to have the same colors as the button:

```
qsetStyleSheet(color = "red", background = "white")
```

Example 11.2: A "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 one is validating entered values. Our implementation uses a stylesheet 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. 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 stye sheets to place an image to the left of the entry message and set the tooltip.

```
qsetMethod("setError", LineEditWithError, function(msg) {
  f <- system.file("images/cancel.gif", package="gWidgets")
  qsetStyleSheet("background-image" = sprintf("url(%s)", f),
                 "background-repeat" = "no-repeat",
```

```
                  "background-position" = "left top",
                  "padding-left" = "20px",
                  widget = this)
  setToolTip(msg)
})
```

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

We can clear the error by resetting the properties to `NULL`.

```
qsetMethod("clearError", LineEditWithError, function() {
  setStyleSheet(NULL)
  setToolTip(NULL)
})
```

11.10 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"` definition through its `load` method:

```
loader <- Qt$QUiLoader()
widget <- loader$load(Qt$QFile("designer.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 one needs to handle signals with R code. There are two ways to accomplish this: manual or automatic.

To manually connect an R handler to a signal, we first need to obtain the widget with the signal. Every widget in a `UI` file is named, so we can call the `qfindChild` utility function to find a specific widget. Assume we have a button named "findButton" and corresponding text entry "findEntry" in our UI file, then we retrieve them with

```
findButton <- qfindChild(widget, "findButton")
findEntry <- qfindChild(widget, "findEntry")
```

Then we connect to the `clicked`QPushButton signal:

```
qconnect(findButton, "clicked", function() {
  findText(findEntry$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$QWidget, function() {
  Qt$QMetaObject$connectSlotsByName(this)
})
```

```
Class 'R::.GlobalEnv::MyMainWindow' with 318 public methods
```

The constructor calls `connectSlotsByName` to automatically establish the connections. 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(findEntry$text)
})
```

In the case of a large, complex GUI, this automatic approach is probably more convenient than manually establishing the connections.

12

Qt: Layout Managers and Containers

Qt provides a set of classes to faciliate 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: the box (`QBoxLayout`), grid (`QGridLayout`) and form (`QFormLayout`). Widgets that function primarily as containers, such as the frame and notebook, are also described here.

Example 12.1: 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 12.1 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:

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
gridLayout <- Qt$QGridLayout()
w$setLayout(gridLayout)
```

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

```
tableWidget <- Qt$QLabel("Table widget")
nbWidget <- Qt$QTabWidget()
buttonLayout <- Qt$QHBoxLayout()
```

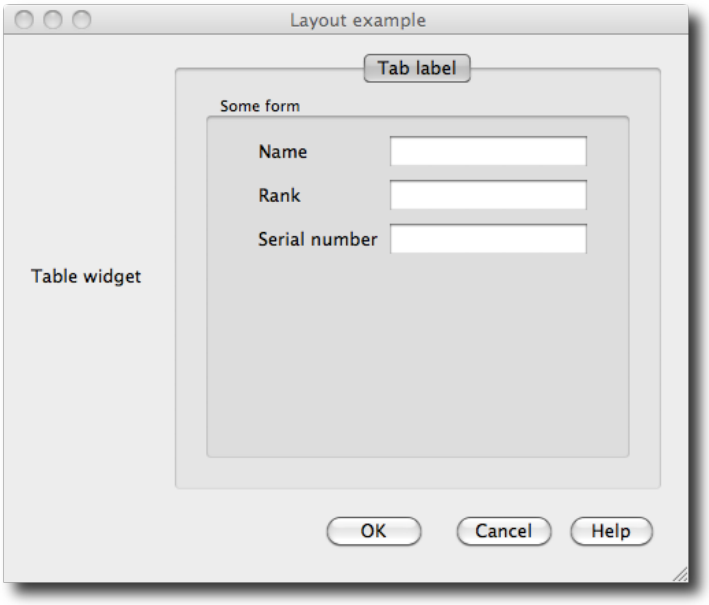

Figure 12.1: A mock GUI illustrating various layout managers provided by Qt.

and add them to the grid layout:

```
gridLayout$addWidget(tableWidget, row=0, column=0,
                     rowspan=1, colspan=1)
gridLayout$addWidget(nbWidget, 0, 1)
gridLayout$addLayout(buttonLayout, 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$QPushButton(i))
buttonLayout$setDirection(Qt$QBoxLayout$RightToLeft)
buttonLayout$addStretch(1L)
buttonLayout$addWidget(b$OK)
buttonLayout$addWidget(b$Cancel)
buttonLayout$addSpacing(12L)
buttonLayout$addWidget(b$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:

```
nbPage <- Qt$QWidget()
```

```
nbWidget$addTab(nbPage, "Tab label")
nbWidget$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:

```
formLayout <- Qt$QFormLayout()
nbPage$setLayout(formLayout)
l <- sapply(c("name", "rank", "snumber"),  Qt$QLineEdit)
formLayout$addRow("Name", l$name)
formLayout$addRow("Rank", l$rank)
formLayout$addRow("Serial number", l$snumber)
```

Each `addRow` call adds a label and an adjacent input widget, in this case a text entry.

This includes 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.

12.1 Layout Basics

Adding and Manipulating Children

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 12.2.

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:

```
wid <- Qt$QWidget()
wid$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, one can nest child layouts by calling `addLayout`.

Internally, layouts store child components as items of class `QLayoutItem`. The item at a given zero-based index is returned by `itemAt`. 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, one 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 well the steps: [1]

1. All the widgets will initially be allocated an amount of space in accordance with their `sizePolicy` and `sizeHint`.
2. If any of the widgets have stretch factors set, with a value greater than zero, then they are allocated space in proportion to their stretch factor.
3. If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an expanding size policy first.
4. Any widgets that are allocated less space than their minimum size (or minimum size hint if no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum size or minimum size hint in which case the strech 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 strech factor is their determining factor.)

Every widget returns a size hint to the layout from the the `sizeHint` method/property. The interpretation of the size hint depends on the `sizePolicy` property. The size policy is an object of class `QSizePolicy`. It contains a separate policy value, taken from the `QSizePolicy` enumeration, for the vertical and horizontal directions. 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 **??**.

[1] `http://doc.qt.nokia.com/4.6/layout.html`

Table 12.1: Possible size policies

Policy	Meaning
Fixed	to require the size hint exactly
Minimum	to treat the size hint as the minimum, allowing expansion
Maximum	to treat the size hint as the maximum, allowing shrinkage
Preferred	to request the size hint, but allow for either expansion or shrinkage
Expanding	to treat like `Preferred`, except the widget desires as much space as possible
MinimumExpanding	to treat like `Minimum`, except the widget desires as much space as possible
Ignored	to ignore the size hint and request as much space as possible

tab:qt:size-policies As an example, consider `QPushButton`. It is the convention that a button will only allow horizontal, but not vertical, expansion. It also requires enough space to display its entire label. Thus a `QPushButton` instance returns a size hint depending on the label dimensions and has the policies `Fixed` and `Minimum` as its vertical and horizontal policies respectively. We could prevent a button from expanding at all:

```
b <- Qt$QPushButton("No expansion")
b$setSizePolicy(vertical=Qt$QSizePolicy$Fixed,
                horizontal=Qt$QSizePolicy$Fixed)
```

Thus, the sizing behavior is largely inherent to the widget or its layout, if any, rather than any parent layout parameters. This is a major difference from GTK+, where a widget can only request a minimum size and all else depends on the parent container widget. The Qt approach seems better at encouraging consistency in the layout behavior of widgets of a particular type.

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:

```
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 andmay 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 5 pixels of space:

```
layout$spacing <- 5L
```

12.2 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:

```
hb <- Qt$QHBoxLayout()
```

Child widgets are added to a box container through the `addWidget` method:

```
buttons <- sapply(letters[1:3], Qt$QPushButton)
sapply(buttons, hb$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. For 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`. 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 may be modified with `setStretchFactor`:

```
hb$setStretchFactor(wid, 2.0)
```

```
[1] FALSE
```

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 12.1:

```
buttonLayout$addSpacing(12L)
```

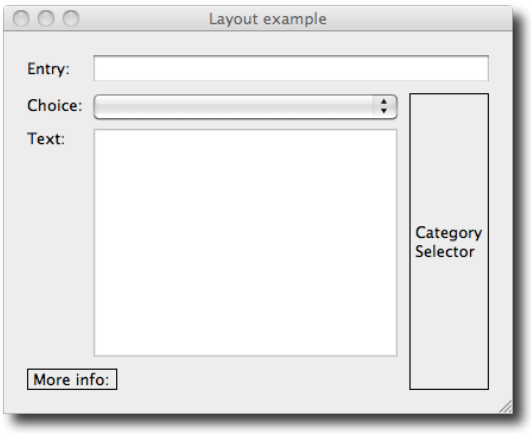

Figure 12.2: A mocked up layout using the `QGridLayout` class.

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:

```
g$addStretch(2)
g$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:

```
g$addStrut(10)                    # atleast 10 pixels high
```

```
NULL
```

12.3 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 12.2). To begin, we create the window with the grid layout:

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
lyt <- Qt$QGridLayout()
w$setLayout(lyt)
```

When we add a child to the grid layout, we need to specify the zero-based row and column indices:

```
lyt$addWidget(Qt$QLabel("Entry:"), 0, 0)
lyt$addWidget(Qt$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:

```
lyt$addWidget(Qt$QLabel("Choice:"), 1, 0)
lyt$addWidget(Qt$QComboBox(), 1, 1)
```

The bottom three cells in the third column are managed by a sublayout, in this case a vertical box layout. We use a label as a stub and set a frame style to have it stand out:

```
lyt$addLayout(slyt <- Qt$QVBoxLayout(), 1, 2, rowspan=3, 1)
slyt$addWidget(l <- Qt$QLabel("Category\nSelector"))
l$setFrameStyle(Qt$QFrame$Box)
```

The text edit widget is added to the third row, second column:

```
lyt$addWidget(Qt$QLabel("Text:"), 2, 0, Qt$Qt$AlignTop)
lyt$addWidget(l <- Qt$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:

```
lyt$addWidget(l <- Qt$QLabel("More info:"), 3, 0,
              rowspan=1, colspan=2)
l$setSizePolicy(Qt$QSizePolicy$Fixed, Qt$QSizePolicy$Preferred)
l$setFrameStyle(Qt$QFrame$Box)
```

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:

```
lyt$setRowStretch(2, 1)                    # third row
lyt$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:

```
lineEdit <- lyt$itemAtPosition(0, 1)$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.

12.4 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:

```
w <- Qt$QWidget()
w$setWindowTitle("Wrapper for 'dnorm' function")
w$setLayout(flyt <- Qt$QFormLayout())
flyt$addRow("quantile", Qt$QLineEdit())
flyt$addRow("mean", Qt$QLineEdit())
flyt$addRow("sd", Qt$QLineEdit())
flyt$addRow(Qt$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 zero-based row index and the role of the desired widget. Here, we obtain the edit box for the quantile parameter:

```
quantileEdit <- flyt$itemAt(0, Qt$QFormLayout$FieldRole)
```

12.5 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 13.5 for an example. The title, stored in the `title` property, may be aligned to left, right or center, depending on the `alignment` property, see Figure **??**. If the `checkable` property is `"TRUE"`, the frame contents can be enabled or disabled by clicking a check button.

12.6 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, one configures 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.

12.7 Notebooks

A notebook container is provided by the class `QTabWidget`:

```
nb <- Qt$QTabWidget()
```

To create a page, one needs to specify the label for the tab and the widget to display when the page is active:

```
nb$addTab(Qt$QPushButton("page 1"), "page 1")
icon <- Qt$QIcon("small-R-logo.jpg")
nb$addTab(Qt$QPushButton("page 2"), icon,  "page 2")
```

As shown in the second call to `addTab`, one can provide an icon to display next to the tab label. We can also add a tooltip for a specific tab, using zero-based indexing:

```
nb$setTabToolTip(0, "This is the first page")
```

The `currentIndex`roperty holds the zero-based index of the active tab. We make the second tab active:

```
nb$currentIndex <- 1
```

The tabs can be positioned on any of the four sides of the notebook; this depends on the `tabPosition`roperty. By default, the tabs are on top, or `"North"`. We move them to the bottom:

```
nb$tabPosition <- Qt$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:

```
nb$tabsClosable <- TRUE
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
nb$movable <- TRUE
nb$usesScrollButtons <- TRUE
```

We need to connect to the `tabCloseRequested` signal to actually close the tab when the close button is clicked.

Example 12.2: 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")
  f <- function(url) openPage(url)
  options("browser" = f)
})
```

```
Class 'R:::.GlobalEnv::HelpBrowser' with 367 public methods
```

The lone method is that called to open page.

```
qsetMethod("openPage", HelpBrowser, function(url) {
  nm <- strsplit(url, "/")[[1]]
  nm <- sprintf("%s: %s", nm[length(nm)-2], nm[length(nm)])
  w <- Qt$QWebView()
  w$setUrl(Qt$QUrl(url))
  i <- addTab(w, nm)
})
```

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

General Widget Stacking It is sometimes useful to have a widget that only shows one of its widgets at once, like a `QTabWidget`, except without the tabs. There is no way to hide the tabs of `QTabWidget`. Instead, one 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.

12.8 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$QLabel()
image$pixmap <- Qt$QPixmap("someimage.png")
sa <- Qt$QScrollArea()
sa$setWidget(image)
```

Next, we define a function for zooming the image:

```
zoomImage <- function(x = 2.0) {
  image$resize(x * image$pixmap$size())
  updateScrollBar <- function(sb) {
    sb$value <- x * sb$value + (x - 1) * sb$pageStep / 2
  }
  updateScrollBar(sa$horizontalScrollBar())
  updateScrollBar(sa$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.

12.9 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 and change the orientation from horizontal to vertical:

```
sp <- Qt$QSplitter()
sp$addWidget(Qt$QLabel("One"))
sp$addWidget(Qt$QLabel("Two"))
sp$addWidget(Qt$QLabel("Three"))
sp$setOrientation(Qt$Qt$Vertical)
```

We can adjust the sizes programmatically:

```
sp$setSizes(c(100L, 200L, 300L))
```

13

Qt: Widgets

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 a view for a separate data model.

13.1 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 provide two ways to create a message box. The simplest approach is to call a static convenience method for issuing common types of messages, including warnings and simple questions. The alternative, described later, involves several steps and offers more control at a cost of convenience. Here we take the simple path for presenting a warning dialog:

```
response <- Qt$QMessageBox$warning(parent = NULL,
                title = "Warning!", text = "Warning message...")
```

This 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

of button corresponds to a fixed type of response. The standard button/response codes are listed in the `QMessageBox::StandardButton` enumeration. In this case, there is only a single button, `"QMessageBox$Ok"`. The dialog is *modal*, meaning that the user cannot interact with the `"parent"` window until responding. If the `"parent"` is `"NULL"`, as in this case, input to all windows is blocked. The dialog is automatically positioned near its parent, and if the parent is destroyed, the dialog is destroyed, as well. Additional arguments specify the available buttons/responses and the default response. We have relied on the default values for these.

For more control over the appearance and behavior of the dialog, we will take a more gradual path. First, we construct an instance of `QMessageBox`. It is possible to specify several properties at construction. Here is how one might construct a warning dialog:

```
dlg <- Qt$QMessageBox(icon = Qt$QMessageBox$Warning,
                      title = "Warning!",
                      text = "Warning text...",
                      buttons = Qt$QMessageBox$Ok,
                      parent = NULL)
```

This call introduces the `icon` property, which is a code from the `QMessageBox::Icon` enumeration and identifies a standard, themeable icon. The icon also implies the message type, just as a button implies a response type. We also need to specify the possible responses with the `"buttons"` argument.

Our dialog is already sufficiently complete to be displayed. However, we have the opportunity to specify further properties. Two of the most useful are `informativeText` and `detailedText`:

```
dlg$informativeText <- "Less important warning information"
dlg$detailedText <- "Extra details most do not care to see"
```

Both provide additional textual information at an increasing level of detail. The `informativeText`QMessageBox will be rendered as secondary to the actual message text. To display the `detailedText`, the user will need to interact with a control in the dialog. An example is a stack trace for the warning.

After specifying the desired properties, the dialog is shown. The approach to showing the dialog depends on whether the dialog should be modal. A modal dialog is displayed with the `exec` method.

```
dlg$exec()
```

```
[1] 1024
```

As its name implies, `exec` executes a loop that will block until the user responds. As with the static convenience methods, the return value indicates the button/response.

To present a non-modal dialog, we first need to register a response listener, as the response will arrive asynchronously:

```
qconnect(dlg, "finished", function(response) {
  ## handle response
  dlg$close()
})
```

```
QObject instance
```

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.

Then we show, raise and activate the dialog:

```
dlg$show()
dlg$raise()
dlg$activateWindow()
```

Modal dialogs may be window modal (`Qt$Qt$WindowModal`), where the dialog blocks all access to its ancestor windows, or application modal (`Qt$Qt$ApplicationModal`) (the default) where all windows are blocked. To specify the type of modality, call `setWindowModality`.

To summarize, we present a general message box, supporting multiple responses:

```
dlg <- Qt$QMessageBox()
dlg$windowTitle <- "[This space for rent]"
dlg$text <- "This is the main text"
dlg$informativeText <- "This should give extra info"
dlg$detailedText <- "And this provides\neven more detail"
dlg$icon <- Qt$QMessageBox$Critical
dlg$standardButtons <- Qt$QMessageBox$Cancel | Qt$QMessageBox$Ok
## 'Cancel' instead of 'Ok' is the default
dlg$setDefaultButton(Qt$QMessageBox$Cancel)
if(dlg$exec() == Qt$QMessageBox$Ok)
  print("A Ok")
```

Input dialogs

The `QInputDialog` class provides a convenient means to gather information from the user and is in a sense the inverse of `QMessageBox`. Possible input modes include selecting a value from a list or entering text or numbers. By default, input dialogs consist of an input control, an icon, and two buttons: “Ok” and “Cancel”.

Like `QMessageBox`, one can display a `QInputDialog` either by calling a static convenience method or by constructing an instance and configuring it before showing it. We demonstrate the former approach for a dialog that requests textual input:

```
text <- Qt$QInputDialog$getText(parent = NULL,
                          title = "Gather text",
                          label = "Enter some text")
```

The return value is the entered string, or `NULL` if the user cancelled the dialog. Additional parameters allow one to specify the initial text and to override the input mode, e.g., for password-style input.

We can also display a dialog for integer input. Here, we ask the user for an even integer between 1 and 10:

```
num <- Qt$QInputDialog$getInt(parent = NULL,
                        title = "Gather integer",
                        label = "Enter an integer from 1 to 10",
                        value = 0, min = 2, max = 10, step = 2)
```

The number is chosen using a bounded spin box. To request a real value, call `Qt$QInputDialog$getDouble` instead.

The final type of input is selecting an option from a list of choices:

```
item <- Qt$QInputDialog$getItem(parent = NULL,
                          title = "Select item",
                          label = "Select a letter",
                          items = LETTERS, current = 17)
```

The dialog contains a combo box filled with the capital letters. The initial choice is 0-based index 17, or the letter "R". The chosen string is returned.

`QInputDialog` has a number of options that cannot be specified via one of the static convenience methods. These option flags are listed in the `QInputDialog$InputDialogOption` enumeration and include hiding the "Ok" and "Cancel" buttons and selecting an item with a list widget instead of a combo box. If such control is necessary, we must explicitly construct a dialog instance, configure it, execute it and retrieve the selected item.

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Select item")
dlg$setLabelText("Select a letter")
dlg$setComboBoxItems(LETTERS)
dlg$setOptions(Qt$QInputDialog$UseListViewForComboBoxItems)
```

```
if (dlg$exec())
  print(dlg$textValue())
```

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

Button boxes

Before discussing custom dialogs, we first introduce the `QDialogButtonBox` utility for arranging dialog buttons in a consistent and cross-platform

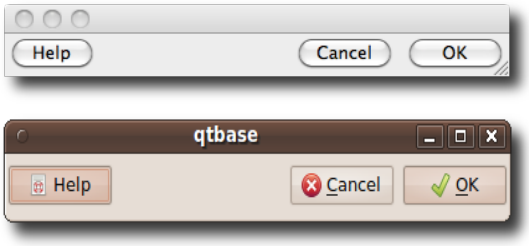

Figure 13.1: Dialog button boxes and their implementation under Mac OS X and Linux.

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:

```
db <- Qt$QDialogButtonBox(Qt$QDialogButtonBox$Ok |
                          Qt$QDialogButtonBox$Cancel |
                          Qt$QDialogButtonBox$Help)
```

Figure 13.1 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, one 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(db, "accepted", function() message("accepted"))
qconnect(db, "rejected", function() message("rejected"))
qconnect(db, "helpRequested", function() message("help"))
qconnect(db, "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`troperty 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 derivates 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$QDialog,
          function(parent = NULL) {
            super(parent=parent)
            setWindowTitle("Choose a date")
            this$calendar <- Qt$QCalendarWidget()
            #
            buttonBox <- Qt$QDialogButtonBox(Qt$QMessageBox$Cancel |
                                             Qt$QMessageBox$Ok)
            qconnect(buttonBox, "accepted", function() {
              this$close()
              this$setResult(Qt$QMessageBox$Ok)
            })
            qconnect(buttonBox, "rejected", function() this$close())
            #
            layout <- Qt$QVBoxLayout()
            sapply(list(calendar, buttonBox), layout$addWidget)
            setLayout(layout)
          })
```

```
Class 'R:::.GlobalEnv::DateDialog' with 336 public methods
```

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())
```

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

`DateDialog` can be executed like any other `QDialog`:

```
dateDialog <- DateDialog()
if (dateDialog$exec())
  message(dateDialog$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 backwards and forwards 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:

```
getAgePage <- Qt$QWizardPage(wizard)
getAgePage$setTitle("What is your age?")
lyt <- Qt$QFormLayout()
getAgePage$setLayout(lyt)
lyt$addRow("Age", (age <- Qt$QLineEdit()))
wizard$addPage(getAgePage)
```

Two more pages are added:

```
getToysPage <- Qt$QWizardPage(wizard)
getToysPage$setTitle("What toys do you like?")
lyt <- Qt$QFormLayout()
getToysPage$setLayout(lyt)
lyt$addRow("Toys", (toys <- Qt$QLineEdit()))
wizard$addPage(getToysPage)
getGamesPage <- Qt$QWizardPage(wizard)
getGamesPage$setTitle("What games do you like?")
lyt <- Qt$QFormLayout()
getGamesPage$setLayout(lyt)
lyt$addRow("Games", (games <- Qt$QLineEdit()))
wizard$addPage(getGamesPage)
```

Finally, we run the wizard by calling its `exec` method:

```
ret <- wizard$exec()
```

File and Directory Choosing Dialogs

`QFileDialog` allows the user to select files and directories, by default using the platform native file dialog. As with other dialogs there are static methods to create dialogs with standard options. These are `"getOpenFileName"`,

"getOpenFileNames", "getExistingDirectory", and "getSaveFileName". To select a file name to open we would have:

```
fname <- Qt$QFileDialog$getOpenFileName(NULL, "Open a file...", getwd())
```

All take as intial arguments a parent, a caption and a directory. Other arguments allow one to set a filter, say. For basic use, these are nearly as easy to use as R's `file.choose`. If a file is selected, `fname` will contain the full path to the file, otherwise it will be `NULL`.

To allow multiple selection, call the plural form of the method:

```
fnames <- Qt$QFileDialog$getOpenFileNames(NULL,
                          "Open file(s)...", getwd())
```

To select a file name for saving, we have

```
fname <- Qt$QFileDialog$getSaveFileName(NULL,
                          "Save as...", getwd())
```

And to choose a directory,

```
dname <- 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:

```
nameFilter <- paste("R files (*.R .RData)",
                    "Sweave files (*.Rnw)",
                    "All files (*.*)",
                    sep=";;")
#
fnames <- Qt$QFileDialog$getOpenFileNames(NULL,
              "Open file(s)...", getwd(), nameFilter)
```

Although the static functions provide most of the functionality, to fully configure a dialog, it may be necessary to explicitly construct and manipulate a dialog instance. Examples of options not available from the static methods are history (previously selected file names), sidebar shortcut URLs, and filters based on low-level file attributes like permissions.

Example 13.1: File dialogs
We construct a dialog for opening an R-related file, using the file names selected above as the history:

```
dlg <- Qt$QFileDialog(NULL, "Choose an R file", getwd(), nameFilter)
dlg$fileMode <- Qt$QFileDialog$ExistingFiles
dlg$setHistory(fnames)
```

The dialog is executed like any other. To get the specified files, call `selectedFiles`:

```
if(dlg$exec())
  print(dlg$selectedFiles())
```

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.

13.2 Labels

As seen in previous example, basic labels in Qt are instances of the `QLabel` class. Labels in Qt are the primary means for displaying static text and images. Textual labels are the most common, and the constructor accepts a string for the text, which can be plain text or, for rich text, HTML. Here we use HTML to display red text:

```
l <- 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 overrides this.

The label text is stored in the `text` property. Properties relevant to text layout include: `alignment`, `indent` (in pixels), `margin`, and `wordWrap`.

13.3 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 may 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.

A push button usually executes some command when clicked, i.e., when the `clicked` signal is emitted:

```
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 may be defined for different sizes and display modes (normal, disabled, active, selected); however, this is often not necessary, as Qt will automatically 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:

```
iconFile <- system.file("images/ok.gif", package="gWidgets")
button$icon <- Qt$QIcon(iconFile)
```

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]. One 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 and plot the icon with `grid`:

```
require(qtutils)
device <- QT()
grid.newpage()
grid.draw(GeomHistogram$icon())
```

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)
```

[1] `QPixmap` is not to be confused with `QImage`, which is optimized for image manipulation, or the vector-based `QPicture`

```
device$render(painter)
painter$end()
```

Finally, we use the icon in a button:

```
b <- Qt$QPushButton("Histogram")
b$setIcon(Qt$QIcon(pixmap))
```

13.4 Checkboxes

The `QCheckBox` class implements a checkbox. Like the `QPushButton` class, `QCheckBox` extends `QAbstractButton`. Thus, `QCheckBox` inherits the signals `clicked`, `pressed`, and `released`. We create a check box for our demonstration:

```
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, one needs 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")
})
```

```
QObject instance
```

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 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, 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:

```
w <- Qt$QGroupBox("Cylinders:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

Next, we create the button group:

```
bg <- Qt$QButtonGroup()
bg$exclusive <- FALSE
```

By default, the buttons are exclusive, like a radio button group. We disable that 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")
cyls <- levels(Cars93$Cylinders)
sapply(seq_along(cyls), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cyls[i]))
  lyt$addWidget(button)
  bg$addButton(button, i)
})
sapply(bg$buttons(), function(i) i$checked <- TRUE)
```

Every button is initially checked.

We can retrieve a list of the buttons in the group and query their checked state:

```
checked <- sapply(bg$buttons(), function(i) i$checked)
if(any(checked)) {
  ind <- Cars93$Cylinders %in% cyls[checked]
  print(sprintf("You've selected %d cases", sum(ind)))
}
```

By attaching a callback to the `buttonClicked` signal, we will be informed when any of the buttons in the group are clicked:

```
qconnect(bg, "buttonClicked", function(button) {
  message(paste("Level '", button$text, "': ", button$checked, sep = ""))
})
```

13.5 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. Continuing our filtering example, we create several radio buttons for choosing a range for the "Weight" variable in the "Cars93" dataset:

```
l <- list(Qt$QRadioButton("Weight < 3000", w),
          Qt$QRadioButton("3000 <= Weight < 4000", w),
          Qt$QRadioButton("4000 <= Weight", w))
```

The simplest way to group the radio boxes is to place them into the same layout:

```
w <- Qt$QGroupBox("Weight:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
sapply(l, function(i) lyt$addWidget(i))
l[[1]]$setChecked(TRUE)
```

As with any other derivative of QAbstractButton, the checked state is stored in the checked property:

```
l[[1]]$checked
```

```
[1] TRUE
```

The toggled signal is emitted twice when a button is checked or unchecked:

```
sapply(l, function(i) {
  qconnect(i, "toggled", function(checked) {
    if(checked) {
      message(sprintf("You checked %s.", i$text))
    }
  })
})
```

QButtonGroup is a useful utility for grouping radio buttons:

```
buttonGroup <- Qt$QButtonGroup()
lapply(l, buttonGroup$addButton)
```

Since our button group is exclusive, we can query for the currently checked button:

```
buttonGroup$checkedButton()
```

```
QRadioButton instance
```

13.6 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 14.8.

This example shows how one combobox, listing regions in the U.S., updates another, which lists states in that region. 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)
statesByRegion <- split(df, df$region)
```

We create our combo boxes, loading the `region` combobox with the regions:

```
state <- Qt$QComboBox()
region <- Qt$QComboBox()
region$addItems(names(statesByRegion))
```

The `addItems` accepts a character vector of options and is the most convenient way to populate a combo box with a simple list of strings. The `currentIndex` property indicates the index of the currently selected item:

```
region$currentIndex
```

```
[1] 0
```

```
region$currentIndex <- -1
```

By setting it to -1, we make the selection to be empty.

To respond to a change in the current index, we connect to the `activated` signal:

```
qconnect(region, "activated", function(ind) {
  state$clear()
  state$addItems(statesByRegion[[ind+1]]$name)
})
```

```
QObject instance
```

Our handler resets the state combo box to correspond to the selected region, indicated by `"ind"`.

Finally, we place the widgets in a form layout:

```
w <- Qt$QGroupBox("Two comboboxes")
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Region:", region)
lyt$addRow("State:", state)
```

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.

13.7 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 spinboxes 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`. `QSlider` selects only from integer values. We create an instance and specify the bounds of the range:

```
sl <- Qt$QSlider()
sl$minimum <- 0
sl$maximum <- 100
```

We can also customize the step size:

```
sl$singleStep <- 1
sl$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 give by the property `value`; we set it to the middle of the range:

```
sl$value
```

```
[1] 0
```

```
sl$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 10 values:

```
sl$orientation <- Qt$Qt$Horizontal
sl$tickPosition <- Qt$QSlider$TicksBelow
sl$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`:

```
sp <- Qt$QSpinBox()
sp$minimum <- sl$minimum
sp$maximum <- sl$maximum
sp$singleStep <- sl$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:

```
sp$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(sp, "valueChanged", f, user.data=sl)
qconnect(sl, "valueChanged", f, user.data=sp)
```

We pass the other widget as the user data, so that state changes in one are forwarded to the other.

13.8 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:

```
le <- Qt$QLineEdit("Initial Contents")
```

The `text`roperty holds the current value:

```
le$text
```

```
[1] "Initial Contents"
```

We wish to select the text, so that the initial contents are overwritten when the user begins typing:

```
le$setSelection(start = 0, length = nchar(le$text))
```

```
NULL
```

```
le$selectedText
```

```
[1] "Initial Contents"
```

If dragEnabled is TRUE the selected text may be dragged and dropped on the appropriate targets. The selectionChanged signal reports selection changes.

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, one can specify place holder text that fills the entry it is empty and unfocused. Typically, this text indicates the expected contents of the entry:

```
le$text <- ""
le$placeholderText <- "Enter some text"
```

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(le, "editingFinished", function() {
  message("Entered text: '", le$text, "'")
})
```

```
QObject instance
```

To respond to any editing, without waiting for it to be committed, connect to the textEdited signal.

Completion

Using the QCompleter framework, a list of possible words can be presented for completion when text is entered into a QLineEdit.

Example 13.2: Completing on 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.

```
w <- Qt$QWidget()
lyt <- Qt$QFormLayout()
w$setLayout(lyt)
lyt$addRow("Class name", c_name <- Qt$QLineEdit())
lyt$addRow("Method name", m_name <- Qt$QLineEdit())
```

Next, we construct the completer for the class entry, listing the components of the "Qt" environment with ls:

```
c_comp <- Qt$QCompleter(ls(Qt))
c_name$setCompleter(c_comp)
```

The completion for the methods depends on the class. As such, we update the completion when editing is finished for the class name:

```
qconnect(c_name, "editingFinished", function() {
  cl <- c_name$text
  if(cl == "") return()
  val <- get(cl, envir=Qt)
  if(!is.null(val)) {
    m_comp <- Qt$QCompleter(ls(val()))
    m_name$setCompleter(m_comp)
  }
})
```

Masks and Validation

`QLineEdit` has various means to restrict and validate user input. The `maxLength` property 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 U.S. Social Security Number:

```
le$inputMask <- "999-99-9999"
```

Please see the API documentation of `QLineEdit` for a full description of the format of an input mask.

As illustrated in Example 11.2, `QValidator` is a much more general validation mechanism, where the value in the widget is checked by the validator before being committed.

13.9 Web View 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"))
```

```
NULL
```

A web browser typically provides feedback on the URL loading process. The signals `loadStartedQWebView`, `loadProgressQWebView` and `loadFinishedQWebView` 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.

13.10 Embedding R Graphics

The `qtutils` package includes a Qt-based graphics device, written by Deepayan Sarkar. We make a simple scatterplot:

```
library(qtutils)
qtDevice <- 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(qtDevice, "Plot 1")
```

```
[1] 0
```

```
print(notebook)
```

```
QTabWidget instance
```

The device provides a context menu with actions for zooming, exporting and printing the plot. One could 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:

```
qtOpenGLDevice <- 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 13.3 for a way to render the scene to an off-screen `QPixmap` for use as an icon.

13.11 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.

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 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(e) {
  md <- Qt$QMimeData()
  md$setText(text)

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)

  drag$exec()
})
```

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

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)
})
```

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

```
qsetMethod("setText", DropLabel, function(str) {
  this$orig_text <- str
  super("setText", str)                 # next method
})
```

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

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(e) {
  md <- e$mimeData()
  if (e$hasImage() || e$hasHtml() | e$hasText()) {
    super("setText", "<Drop Text Here>")
```

```
    setBackgroundRole(Qt$QPalette$Highlight)
    e$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(e) {
  clear()
})
```

Finally, we have the important drop event handler. The following code implements this more generally than is needed for this example, as we only have text in our mime data:

```
qsetMethod("dropEvent", DropLabel, function(e) {
  md <- e$mimeData()

  if(md$hasImage()) {
    setPixmap(md$imageData())
  } else if(md$hasHtml()) {
    setText(md$html)
    setTextFormat(Qt$Qt$RichText)
  } else if(md$hasText()) {
    setText(md$text())
    setTextFormat(Qt$Qt$PlainText)
  } else {
    setText("No match")                  # replace ...
  }

  setBackgroundRole(this$bgrole)
  e$acceptProposedAction()
})
```

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 DnD operation is completed.

14

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 `qtbase`.

14.1 Display of tabular data

Displaying an R data frame

As mentioned, Qt expects data to be stored in a `QAbstractItemModel`. 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 simpler than with GTK+ and `RGtkDataFrame`. Here we show a `data.frame` in a table view:

```
data(mtcars)
model <- qdataFrameModel(mtcars)
view <- Qt$QTableView()
view$setModel(model)
```

```
NULL
```

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 `DataFrameModel` object is a reference, so any changes are reflected in all of its views. The R data frame of a `DataFrameModel` may be accessed using `qdataFrame`:

```
head(qdataFrame(model), 3)
```

```
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4
4
Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4
4
Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4
1
```

```
qdataFrame(model)$hpToMpg <- with(qdataFrame(model), hp / mpg)
```

Our table view now contains a new column, holding the horsepower to miles per gallon ratio. It is important to notice that the view has been updated implicitly, through manipulation of the underlying model.

A view has a horizontal and vertical header. The horizontal header displays the column names, while the vertical header displays the row names, if any. `QHeaderView` is the widget responsible for displaying headers. It has a number of parameters, such as whether the column may 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$AlignLeft
```

`QTableView` provides a number of aesthetic features. By default, a grid is drawn that delineates the cells. One 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., it goes out of scope, the model will be garbage collected, from lack of an owner. For example, this does not work:

```
brokenView <- Qt$QTableView()
brokenView$setModel(qdataFrameModel(mtcars))
```

```
NULL
```

```
gc()
```

```
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 318729 17.1     531268 28.4   467875   25
Vcells 292709  2.3    1597524 12.2  1964309   15
```

```
brokenView$model()
```

```
NULL
```

To prevent this, one 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:

```
parentalView <- Qt$QTableView()
brokenView$setModel(qdataFrameModel(mtcars, parent = parentalView))
```

```
NULL
```

```
gc()
```

```
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 318714 17.1     597831 32.0   467875   25
Vcells 292710  2.3    1278019  9.8  1964309   15
```

```
brokenView$model()
```

```
DataFrameModel instance
```

Formatting cells

Let us now assume that a missing value (`NA`) has been introduced into our dataset:

```
qdataFrame(model)$mpg[1] <- NA
```

The table view will display this as "nan" or "inf", which is inconsitent 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, 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, one creates an instance and sets it as the item delegate for the view:

```
delegate <- qrTextFormattingDelegate()
view$setItemDelegate(delegate)
```

```
NULL
```

Delegates may 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)
wid <- Qt$QWidget()
wid$resize(1000, 500)
vbox <- Qt$QVBoxLayout()
vbox$addWidget(view)
wid$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 from 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 the same initial size (except for the last):

```
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, one wishes 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()
```

```
NULL
```

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). 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 `setResizeModeethod` changes the resize mode of a specific column. Below, we make all of our columns expand:

```
header$resizeMode(Qt$QHeaderView$Stretch)
```

```
Enum value:  (0)
```

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 ellipisizing would be necessary, it may be preferable to instead reduce the widths of the other columns. This mode is enabled with

```
header$cascadingSectionResizes <- TRUE
```

14.2 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 quickly display the first column from a data frame (or anything coercible into a data frame):

```
model <- qdataFrameModel(rownames(mtcars))
view <- Qt$QListView()
view$setModel(model)
```

```
NULL
```

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)
tableView <- Qt$QTableView()
tableView$setModel(model)
```

```
NULL
```

```
listView <- Qt$QListView()
listView$setModel(model)
```

```
NULL
```

Now, when we resort the model, both views will be updated:

```
df <- qdataFrame(model)
qdataFrame(model) <- df[order(df$mpg),]
```

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$QStringListModel(rownames(mtcars))
listView <- Qt$QListView()
listView$setModel(model)
```

```
NULL
```

Now we can retrieve the values as a character vector, rather than as a data frame:

```
head(model$stringList)
```

```
1 function (...)
2 qinvoke(<environment>, "stringList", ...)
```

`QListView` supports features beyond those of a simple list, including features often found in file browsers and desktops. For example, items may be wrapped into additional columns, and an icon mode, supporting unrestricted layout and drag and drop, is also available.

14.3 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$index(0, 0)
index$row()
```

```
[1] 0
```

```
index$column()
```

```
[1] 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:

```
firstCar <- index$data()
```

This can be extended to retrieve all of the items in the list:

```
sapply(seq(model$rowCount()), function(i) model$index(i - 1, 0)$data())
```

```
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
```

```
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
```

Setting the data is also possible, yet requires calling `setData` on the model, not the index:

```
model$setData(index, toupper(firstCar))
```

```
[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 only necessary to directly manipulate a `QModelIndex` when handling user input, as we describe in the next section.

14.4 Item Selection

Selection is likely the most common type of user interaction with lists and tables. The selection state is stored in its own data model, `QItemSelectionModel`:

```
selModel <- listView$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.

There are five selection modes for item views: single, extended, contiguous, multi, and none. These values are defined by the `QAbstractItemView::SelectionMode` enumeration. `"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, and clicking with the Ctrl key pressed adds arbitrary items to the selection. The `"ContiguousSelection"` mode disallows the Ctrl key behavior. To allow selection on mouse-over, with range selection by clicking and dragging, choose `"MultiSelection"`. We configure our list view for single selection:

```
listView$selectionMode <- Qt$QAbstractItemView$SingleSelection
```

```
NULL
```

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 <- selModel$selectedIndexes()
indices[[1]]$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 the selection from the selection model:

```
selection <- selModel$selection()
```

Next, we coerce the `QItemSelection` to an explicit list of `QItemSelectionRange` objects and generate a vector of the selected indices:

```
indicesForSelection <- function(selection) {
  selRanges <- as.list(selection)
  unlist(lapply(selRanges, function(range) seq(range$top(), range$bottom())))
}
indicesForSelection(selection)
```

```
 [1]  2  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 may be wide. Looping over the ranges, but not the individual indices, will be significantly more efficient for large selections.

It is also possible to programmatically change the selection. For example, we may wish to select the first list item:

```
listView$setCurrentIndex(model$index(0, 0))
```

```
NULL
```

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:

```
selModel$select(model$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:

```
selModel$select(model$index(0, 0), Qt$QItemSelectionModel$Deselect)
```

To efficiently select a range of items, we construct a `QItemSelection` object and set it on the model: We have selected items 3 to 10. Multiple ranges may be added to the `QItemSelection` object by calling its `select` method.

For tabular views, selection may be row-wise, column-wise or item-wise (GTK+ supports only row-wise selection). By default, selection is by item. While this is common in spreadsheets, one usually desires row-wise selection in a table, so we will override the default:

```
tableView$selectionBehavior <- Qt$QAbstractItemView$SelectRows
```

Querying a selection 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 the third row (2):

```
selModel <- tableView$selectionModel()
sapply(selModel$selectedRows(), qinvoke, "row")
```

```
list()
```

We invoke the `row` method on each returned `QModelIndex` object to get the row indices. When setting the selection, there are conveniences for selecting an entire row or column. We select the first row of the table:

```
tableView$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:

```
selModel$select(sel, Qt$QItemSelectionModel$Select |
                Qt$QItemSelectionModel$Rows)
```

To respond to a change in selection, connect to the `selectionChanged` signal on the selection model:

```
selectedIndices <- rep(FALSE, nrow(mtcars))
selectionChangedHandler <- function(selected, deselected) {
  selectedIndices[indicesForSelection(selected)] <<- TRUE
  selectedIndices[indicesForSelection(deselected)] <<- FALSE
}
qconnect(selModel, "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.

14.5 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`. After constructing an instance and specifying the child model, the proxy model may be handed to a view like any other model:

```
proxy <- Qt$QSortFilterProxyModel()
proxy$setSourceModel(model)
tableView$setModel(proxy)
listView$setModel(proxy)
```

Our views will now draw data through the proxy, rather than from the original model.

Both table and tree views provide an interface 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:

```
tableView$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$sort(1)
```

The built-in sorting logic understands basic data types like 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, wildcards (globs), and regular expressions. For example, we can filter for cars made by Mercedes:

```
proxy$filterKeyColumn <- 0
proxy$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:

```
tableView$setColumnHidden(4, 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.

14.6 Decorating Items

Thus far, we have only considered 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.

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` supports role-specific values for each item, `"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 model 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, 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)
qdataFrame(model)$.makeAndModel.background <- list(qcolor("gray"))
```

In the above, we store a list of QColor instances in our data frame. As a side note, 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.

The set of supported data types for each role depends on the delegate. For delegates derived from QStyledItemDelegate, see "qhelp(QStyledItemDelegate)". 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.

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))
```

As shown, periods separate the data frame column names in the "NAME" component. To apply a column to all columns in the model, omit the column name:

```
qdataFrame(model)$.font <- list(qfont(pointsize = 14))
```

For models other than DataFrameModel, one sets data for a specific role by passing the optional role argument to setData. The value of role defaults to "EditRole", meaning that the data is in an editable form. We create a list view and set the background of the first item to yellow:

```
listModel <- Qt$QStringListModel(rownames(mtcars))
listModel$setData(listModel$index(0, 0), "yellow", Qt$Qt$BackgroundRole)
listView <- Qt$QListView()
listView$setModel(listModel)
```

14.7 Displaying Hierarchical Data

Hierarchical data is generally stored in QStandardItemModel, the primary implementation of QAbstractItemModel built into Qt. Hierarchical data often arises when splitting a tabular dataset by some combination of factors. For our demonstration, we will display in a tree the result of splitting the Cars93 dataset by manufacturer. The first step of our demonstration is to create the model, with a single column:

```
treeModel <- 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) {
```

Table 14.1: 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)

```
  treeModel$insertRow(treeModel$rowCount())
  manufacturer <- treeModel$index(treeModel$rowCount()-1L, 0)
  treeModel$setData(manufacturer, df$Manufacturer[1])
  treeModel$insertRows(0, nrow(df), manufacturer)
  treeModel$insertColumn(0, manufacturer)
  for (i in seq_along(df$Model)) {
    record <- treeModel$index(i-1L, 0, manufacturer)
    treeModel$setData(record, df$Model[i])
  }
})
```

As before, we need to create a QModelIndex object for accessing each cell of the model. We need to add rows and columns to each manufacturer node before creating its children. This nested loop approach to populating a model is much less efficient than converting a data.frame to a DataFrameModel, but it is necessary to communicate the hierarchical information.

In addition to implementing the QAbstractItemModel interface, QStandardItemModel also represents an item as a QStandardItem object. Many operations, including inserting, removing and manipulating children, may 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 becomes:

```
by(Cars93, Cars93$Manufacturer, function(df) {
  manufacturer <- Qt$QStandardItem(as.character(df$Manufacturer[1]))
  treeModel$appendRow(manufacturer)
  children <- lapply(as.character(df$Model), Qt$QStandardItem)
  lapply(children, manufacturer$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:

```
treeView <- Qt$QTreeView()
treeView$setModel(treeModel)
```

Often, as in our case, a tree view only has a single column. It may be desirable to hide that column header with

```
treeView$headerHidden <- TRUE
```

Columns in a QStandardItemModel may be named by calling setHorizontalHeaderNames, as shown in the next example.

Example 14.1: A workspace browser

This example shows how to use the tree widget item to display a snapshot of the current workspace. Each object in the workspace maps to an item, where recursive objects with names will have their components represented in a hierarchical manner.

When representing objects in a workspace, we need to decide if an object has been changed. To do this, we use the digest function from the digest package to compare a current object with a past one. To store this information, we will use a custom "digest" role in the item model.

```
library(digest)
.DIGEST_ROLE <- Qt$Qt$UserRole + 1L
```

Using a custom role in this manner is convenient but dangerous: third-party code could attempt to store a differenet type of data using the same role ID. It is thus important to document any reserved roles.

The 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, parentObj, parentItem) {
  obj <- parentObj[[varname]]

  item <- Qt$QStandardItem(varname)
  item$setData(digest(obj), .DIGEST_ROLE)
  classItem <- Qt$QStandardItem(paste(class(obj), collapse = ", "))
  print(class(item))
  print(class(classItem))

  parentItem$appendRow(list(item, classItem))

  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, parentItem = item, parentObj = obj)
}
```

Our main function is one that checks the current workspace and updates the values in the tree widget accordingly. This could be set on a timer to be called periodically, or called in response to user input. We consider three cases: items no longer in the workspace to remove, new items to add, and finally items that may have changed and need to be replaced.

```
updateTopLevelItems <- function(view, env = .GlobalEnv) {
  envNames <- ls(envir=env)

  model <- view$model()
  items <- lapply(seq_len(model$rowCount()), model$item, column = 0)
  curNames <- as.character(sapply(items, qinvoke, "text"))



  maybeSame <- curNames %in% envNames

  curDigests <- sapply(items[maybeSame], qinvoke, "data", .DIGEST_ROLE)
  envDigests <- sapply(mget(curNames[maybeSame], env), digest)
  same <- as.character(curDigests) == as.character(envDigests)


  view$updatesEnabled <- FALSE

  remove <- !maybeSame
  remove[maybeSame] <- !same
  sapply(sort(which(remove)-1L, decreasing=TRUE), model$removeRow)

  replaceNames <- curNames[maybeSame][!same]
  newNames <- setdiff(envNames, curNames)

  sapply(c(replaceNames, newNames), addItem, parentObj = env,
         parentItem = model$invisibleRootItem())

  model$sort(0, Qt$Qt$AscendingOrder)
  view$updatesEnabled <- TRUE
}
```

First, we obtain the names and digests for each top-level row. Then, the digests are compared. Names that no longer exist in the environment or have mismatching digests are removed. We need to sort the indices in decreasing order so as not invalidate any indices. The names for the changed objects are then readded, before we add the new names. Finally, we sort the model. While the model is being modified, we freeze the view.

Finally, we construct the model and view:

```
model <- Qt$QStandardItemModel(rows = 0, columns = 2)
model$setHorizontalHeaderLabels(c("Name", "Class"))
view <- Qt$QTreeView()
view$headerHidden <- FALSE
view$setModel(model)
```

This last call initializes the display:

```
updateTopLevelItems(view)
```

14.8 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 use cases; however, it is beneficial to understand that a combo box displays its popup 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:

```
comboBox <- Qt$QComboBox()
comboBox$setModel(model)
```

```
NULL
```

By default, the first column from the model is displayed; this is controlled by the `modelColumn` property.

14.9 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:

```
(treeModel$index(0, 0)$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. We prefix `edit` to the column name, so that the user can change its value between `TRUE` and `FALSE`:

```
df <- mtcars
df$Analyze.edit <- TRUE
model <- qdataFrameModel(df)
```

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.

14.10 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 visually indicate where the item will be dropped. The following enables moving items within a view, i.e., reordering:

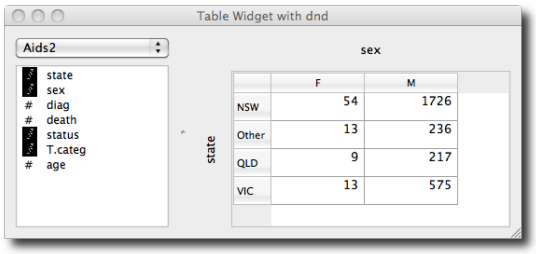

Figure 14.1: A table widget to display contingency tables and a means to specify the variables through drag and drop.

```
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 may be queried with `supportedDragActions` and `supportedDropActions`. The item flags determine whether an individual item may be dragged or dropped upon. Most of the built-in models will support both copy and move actions, when dragging or dropping. `DataFrameModel` only supports copy actions when dragging; dropping is not supported.

Example 14.2: A drag and drop interface to `xtabs`

This examples 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.

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$QWidget, function(parent=NULL) {
  super(parent)
  ## widgets
  this$dfcb <- Qt$QComboBox()
  this$varList <- Qt$QListView()
  this$varList$setModel(qdataFrameModel(data.frame(), this))
  this$varList$dragEnabled <- TRUE

  ## layout
  lyt <- Qt$QVBoxLayout()
  lyt$addWidget(dfcb)
```

```
  lyt$addWidget(varList)
  varList$setSizePolicy(Qt$QSizePolicy$Expanding, Qt$QSizePolicy$Expanding)
  setLayout(lyt)

  updateDataSets()
  qconnect(dfcb, "activated", function(ind) {
    this$dataFrame <- dfcb$currentText
  })
})
```

This utility populates the combo box with a list of data frames:

```
qsetMethod("updateDataSets", VariableSelector, function() {
  curVal <- this$dfcb$currentText
  this$dfcb$clear()
  dfs <- names(which(unlist(eapply(.GlobalEnv, is.data.frame))))
  if(length(dfs)) {
    this$dfcb$addItems(dfs)
    if(is.null(curVal) || !curVal %in% dfs) {
      this$dfcb$currentIndex <- -1
      this$dataFrame <- NULL
    } else {
      this$dfcb$currentIndex <- which(curVal == dfs)
      this$dataFrame <- curVal
    }
  }
})
```

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

The data frame is stored in the following property:

```
qsetProperty("dataFrame", VariableSelector, write = function(df) {
  if (is.null(df))
    df <- data.frame()
  else if (is.character(df))
    df <- get(dfname, .GlobalEnv)
  model <- this$varList$model()
  qdataFrame(model) <- data.frame(variable = names(df),
                                  variable.decoration = I(lapply(df, getIcon)))
  this$.dataFrame <- df
  dataFrameChanged()
})
```

When the property is written, the variable list is updated and it emits this signal:

```
qsetSignal("dataFrameChanged", VariableSelector)
```

The `getIcon` generic resolves an icon from the class of a column:

```
getIcon <- function(x) UseMethod("getIcon")
getIcon.default <- function(x)
  Qt$QIcon(system.file("images/numeric.gif", package="gWidgets"))
getIcon.factor <- function(x)
  Qt$QIcon(system.file("images/factor.gif", package="gWidgets"))
getIcon.character <- function(x)
  Qt$QIcon(system.file("images/character.gif", package="gWidgets"))
```

Next, a derivative of `QLabel` is defined that accepts drops from the variable list and is capable of rotating text for displaying the Y 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("variableName", 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(md) {
  name <- NULL
  RDA_MIME_TYPE <- "application/x-rlang-transport"
  if(md$hasFormat(RDA_MIME_TYPE)) {
    list <- unserialize(md$data(RDA_MIME_TYPE))
    if (length(list) && is.character(list[[1]]))
      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(e) {
  md <- e$mimeData()
  if(!is.null(variableNameFromMimeData(md))) {
    setForegroundRole(Qt$QPalette$Dark)
    e$acceptProposedAction()
  }
```

```
})
qsetMethod("dragLeaveEvent", VariableLabel, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  e$accept()
})
```

To handle the drop, we get the variable name, set the text of the label and emit the variableNameDroppedVariableLabel signal:

```
qsetMethod("dropEvent", VariableLabel, function(e) {
  setForegroundRole(Qt$QPalette$WindowText)
  md <- e$mimeData()
  if(!is.null(this$variableName <- variableNameFromMimeData(md))) {
    this$text <- variableName
    variableNameDropped()
    setBackgroundRole(Qt$QPalette$Window)
    e$acceptProposedAction()
  }
})
```

To complete the VariableLabel class, we override the paintEvent event to respect the rotation class. 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(e) {
  p <- Qt$QPainter()
  p$begin(this)
  w <- this$width; h <- this$height
  p$save()
  p$translate(w/2, h/2)
  p$rotate(-(this$rotation))
  rect <- p$boundingRect(0, 0, 0, 0, Qt$Qt$AlignCenter, this$text)
  p$drawText(rect, Qt$Qt$AlignCenter, this$text)
  p$restore()
  p$end()
})
```

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() {
  ## make Widgets
  this$xlabel <- VariableLabel()
  qconnect(xlabel, "variableNameDropped", invokeXtabs)

  this$ylabel <- VariableLabel()
  pt <- this$ylabel$font$pointSize()
  this$ylabel$minimumWidth <- 2*pt; this$ylabel$maximumWidth <- 2*pt
  this$ylabel$rotation <- 90L
  qconnect(ylabel, "variableNameDropped", invokeXtabs)

  this$tableView <- Qt$QTableView()
  this$tableView$setModel(qdataFrameModel(data.frame(), this))
  clearLabels()
})
```

The `xlabel` is straight-forward: we construct it 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:

```
qsetMethod("invokeXtabs", XtabsWidget, function() {
  if (is.null(dataFrame))
    return()
  if(is.null(xVar <- this$xlabel$variableName)) {
    out <- NULL
  } else if(is.null(yVar <- this$ylabel$variableName)) {
    f <- formula(sprintf("~ %s", xVar))
    out <- xtabs(f, data=get(dataFrame))
  } else {
    f <- formula(sprintf("~ %s + %s", yVar, xVar))
    out <- xtabs(f, data=get(dataFrame))
  }
  if(!is.null(out))
    updateTableView(out)
})
```

We define a method to update the table view:

```
qsetMethod("updateTableView", XtabsWidget, function(table) {
  model <- this$tableView$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 on `XtabsWidget`:

```
qsetProperty("dataFrame", XtabsWidget, write = function(df) {
  clearLabels()
  this$.dataFrame <- df
})
```

All that remains is to place the `VariableSelector` and `XtabsWidget` together in a split pane and then connect a handler that keeps the datasets 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()
```

Figure 14.1 shows the result, after the user has dragged two variables onto the labels.

14.11 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. As with combo boxes, we can populate the items directly from a character vector through the `addItems` method:

```
listWidget <- Qt$QListWidget()
listWidget$addItems(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", listWidget)
```

The first argument is the text and the optional second argument a parent `QListWidget`. If no parent is specified, the the item may be added through the methods `addItem`, or `insertItem` for inserting to a specific instance.

To retrieve an item given its index, we call the `item` method:

```
first <- listWidget$item(0)
first$text()
```

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

Many aspects of an item may be manipulated. These roughly correspond to the built-in roles of items in `QAbstractItemModel`. One may specify the text, font, icon, status and tool tips, and 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:

```
listWidget$selectionMode <- Qt$QListWidget$ExtendedSelection
```

We can programmatically select the states that begin with "A":

```
sapply(grep("^A", state.name),
       function(i) listWidget$item(i - 1)$setSelected(TRUE))
```

The method `selectedItems` will return the selected items in a list:

```
selected <- listWidget$selectedItems()
sapply(selected, qinvoke, "text")
```

```
[1] "Alabama"  "Alaska"   "Arizona"  "Arkansas"
```

To handle changes in the selection, connect to `itemSelectionChanged`:

```
qconnect(listWidget, "itemSelectionChanged", function() {
  selected <- listWidget$selectedItems()
  selectedText <- sapply(selected, qinvoke, "text")
  message("Selected: ", paste(selectedText, collapse = ", "))
})
```

```
QObject instance
```

It is often easier for the user to select multiple items by clicking a check button next to the desired items. The check box is only shown if we explicitly set the check state of item. The possible values are `"Checked"`, `"Unchecked"` or `"PartiallyChecked"`. Here, we set all of the items to unchecked to show the check buttons and then check the selected items:

```
items <- sapply(seq(listWidget$count), function(i) {
  listWidget$item(i - 1)$setCheckState(Qt$Qt$Unchecked)
})
sapply(selected, function(x) x$setCheckState(Qt$Qt$CheckedState))
```

For long lists, this looping will be time consuming. In such cases, it is likely preferable use `QListView`, `DataFrameModel` and the `"CheckedStateRole"`.

14.12 Implementing Custom Models

Normally, the `DataFrameModel` and the models in Qt are sufficient. One can imagine other cases, however. For example, one 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 implementation of a model must provide the methods `rowCount`, `columnCount`, and `data`. The first two describe the size of the table for any views, and the third describes provides data to the view for a particular cell and role. We have already demonstrated the use of the `data` method in the previous sections. For example, if one is 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. One may also implement the `headerData` method to populate the view headers.

Editable Models For editable models, one must implement the `flags` method to return a flag containing `ItemIsEditable` and the `setData` method.

When a value is updated, one should call the `dataChanged` method to notify the views that a portion of the model is changed. This method takes two indices, together specifying a rectangle in the table.

To provide for resizable tables, `Qt` requires one to 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 14.3: 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 no reason to actually 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 Qtviews call far too often.

Our basic constructor simply creates a `dataframe` property and sets the data frame:

```
qsetClass("DfModel", Qt$QAbstractTableModel,
          function(dataframe=data.frame(V1=character(0)), parent=NULL) {
            super(parent)
            this$dataframe <- dataframe
          })
```

Next, we define the `dataframe` property. When a new data frame is set, we call the `dataChanged` method to notify any views of a change:

```
qsetProperty("dataframe", DfModel, write = function(df) {
  this$.dataframe <- df
  dataChanged(index(0, 0), index(nrow(df), ncol(df)))
})
```

There are three virtual methods that we are required to implement: `rowCount`, `columnCount` and `data`. The first two simply access the dimensions of the data frame:

```
qsetMethod("rowCount", DfModel, function(index) nrow(this$dataframe) + 1)
qsetMethod("columnCount", DfModel, function(index) ncol(this$dataframe))
```

The `data` method is the main method to implement. We wish to customize the data display based on the class of the variable represented in a column. We implement this with S3 methods, and several are defined below:

```
displayRole <- function(x, row, ...) UseMethod("displayRole")
displayRole.default <- function(x, row)
  sprintf("%s", x[row])
displayRole.numeric <- function(x, row)
```

```
  sprintf("%.2f", x[row])
displayRole.integer <- function(x, row)
  sprintf("%d", x[row])
```

We see that numeric values are formatted to have 2 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) {
  d <- this$dataframe
  row <- index$row()
  col <- index$column() + 1

  if(role == Qt$Qt$DisplayRole) {
    if(row > 0)
      displayRole(d[,col], row)
    else
      names(d)[col]
  } else if(role == Qt$Qt$EditRole) {
    if(row > 0)
      as.character(d[row, col])
    else
      names(d)[col]
  } else {
    NULL
  }
})
```

To allow the user to edit the values we need to override the `flags` method to return `ItemIsEditable` in the flag, so that any views are aware of this ability:

```
qsetMethod("flags", DfModel, function(index) {
  if(!index$isValid()) {
    return(Qt$Qt$ItemIsEnabled)
  } else {
    curFlags <- super("flags", index)
    return(curFlags | Qt$Qt$ItemIsEditable)
  }
})
```

To edit cells we need to implement a method to set data once edited. 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 `fitIn`, which will coerce the string to the original type. For example:

```
fitIn <- function(x, value) UseMethod("fitIn")
fitIn.default <- function(x, value) value
fitIn.numeric <- function(x, value) as.numeric(value)
```

The `setData` method is responsible for taking the value from the delegate and assigning it into the model:

```
qsetMethod("setData", DfModel, function(index, value, role) {
  if(index$isValid() && role == Qt$Qt$EditRole) {
    d <- this$dataframe
    row <- index$row()
    col <- index$column() + 1

    if(row > 0) {
      x <- d[, col]
      d[row, col] <- fitIn(x, value)
    } else {
      names(d)[col] <- value
    }
    this$dataframe <- d
    dataChanged(index, index)

    return(TRUE)
  } else {
     super("setData", index, value, role)
  }
})
```

For a data frame editor, we may wish to extend the API for our table of items to be R specific. For example, this method allows one to replace a column of values:

```
qsetMethod("setColumn", DfModel, function(col, value) {
  ## pad with NA if needed
  n <- nrow(this$dataframe)
  if(length(value) < n)
    value <- c(value, rep(NA, n - length(value)))
  value <- value[1:n]
  d <- this$dataframe
  d[,col] <- value
  this$dataframe <- d          # only notify about this column
  dataChanged(index(0, col-1), index(rowCount()-1, col-1))
  return(TRUE)
})
```

We implement a method similar to the `insertColumn` method, but specific to our task. Since we may add a new column, we call the "begin" and "end" methods to notify any views.

```
qsetMethod("addColumn", DfModel, function(name, value) {
  d <- this$dataframe
  if(name %in% names(d)) {
    return(setColumn(min(which(name == names(d))), value))
  }
  beginInsertColumns(Qt$QModelIndex(), columnCount(), columnCount())
  d[[name]] <- value
  this$dataframe <- d
  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:

```
triggerFlag <- Qt$QAbstractItemView$DoubleClicked |
               Qt$QAbstractItemView$SelectedClicked |
               Qt$QAbstractItemView$EditKeyPressed
view$setEditTriggers(triggerFlag)
view$verticalHeader()$setHidden(TRUE)
view$horizontalHeader()$setHidden(TRUE)
```

14.13 Alternative Views of Data Models

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* 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 14.4: Mapping selected model items to a text entry
We will demonstrate `QDataWidgetMapper` by displaying a table view of the `"Cars93"` dataset, 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=TRUE)
mapper <- Qt$QDataWidgetMapper()
mapper$setModel(model)
label <- Qt$QLabel()
mapper$addMapping(label, 1)
```

The `addMapping` establishes a mapping between the view widget and the 0-based column index in the model.

Next, we construct a table view and establish a handler that changes the current row of the data mapper upon selection:

```
tableView <- Qt$QTableView()
tableView$setModel(model)
qconnect(tableView$selectionModel(), "currentRowChanged",
         mapper$setCurrentIndex)
```

Finally, we layout our GUI:

```
w <- Qt$QWidget()
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
lyt$addWidget(tableView)
lyt$addWidget(label)
```

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 14.5: 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 display a mean value for a given column. Our custom view class will be a sub-class of `QLabel`. This is a simple illustration, where we provide a label with text summarizing the mean of the values in the first column of the model.

In the constructor we define a label property and call our `setModel` method:

```
qsetClass("MeanLabel", Qt$QLabel, function(model, column = 0, parent=NULL) {
```

```
  super(parent)
  this$model <- model
  this$column <- column
  qconnect(model, "dataChanged", function(topLeft, bottomRight) {
    if (topLeft$column() <= column && bottomRight$column() >= column)
      updateMean()
  })
  updateMean()
})
```

Whenever the data in the model changes, we need to update the display of the mean value. This private method performs the update:

```
qsetMethod("updateMean", MeanLabel, function() {
  if(is.null(model)) {
    txt <- "No model"
  } else {
    df <- qdataFrame(model)
    cname <- colnames(df)[column+1L]
    xbar <- mean(df[,cname])
    txt <- sprintf("Mean for '%s': %s", cname, xbar)
  }
  this$text <- txt
}, 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))
tableView <- Qt$QTableView()
tableView$setModel(model)
tableView$setEditTriggers(Qt$QAbstractItemView$DoubleClicked)
meanLabel <- MeanLabel(model)
w <- Qt$QWidget()
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
lyt$addWidget(tableView)
lyt$addWidget(meanLabel)
```

14.14 Viewing and Editing Text Documents

Multi-line text is displayed and edited by the `QTextEdit` widget, which is the view and controller for the `QTextDocument` model. `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.

Here, we create a QTextEdit and populate it with some text. Although the text is actually stored in a QTextDocument, it is usually sufficient to interact with the QTextEdit directly:

```
te <- Qt$QTextEdit()
te$setPlainText("The quick brown fox")
te$append("jumped over the lazy dog")
```

```
te$toPlainText()
```

```
[1] "The quick brown fox\njumped over the lazy dog"
```

The textChanged signal is emitted when the text is changed.

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. We obtain the user-visible cursor and move it to the end of the document:

```
n <- nchar(te$toPlainText())
cursor <- te$textCursor()
cursor$setPosition(n)
te$setTextCursor(cursor)
```

Manipulating the cursor object does not actually modify the location and parameters of the cursor on the screen. We need to explicitly set the modified cursor object on the QTextEdit. 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 an image at the beginning:

```
cursor$setPosition(0)
cursor$insertImage(system.file("images/ok.gif", package="gWidgets"))
```

To listen to changes in the cursor position, connect to the cursorPositionChanged signal on the QTextEdit.

Selections Selection is a component of the QTextCursor state. For plain text, the selected text is returned by the selectedText method:

```
te$textCursor()$selectedText()
```

```
NULL
```

The NULL value indicates that the user has not selected any text. The selection spans from the cursor anchor to the cursor position. Normally, the anchor and cursor are at the same position. To make a selection, we

move the cursor independently of its anchor. To set the selection to include the first three words of the text, we have:

```
cursor <- Qt$QTextCursor(te$document())
cursor$movePosition(Qt$QTextCursor$Start)
cursor$movePosition(Qt$QTextCursor$WordRight, Qt$QTextCursor$KeepAnchor, 3)
te$setTextCursor(cursor)
```

```
cursor$selectedText()
```

```
[1] "ï£ijThe quick brown "
```

We move the cursor and anchor to the start of the document. Next, we move the cursor, without the anchor, across the right end of three words. Finally, we need to commit the modified cursor.

To listen to changes in the selection (according to the user visible cursor), connect to `selectionChanged`:

```
qconnect(te, "selectionChanged", function() {
  message("Selected text: '", te$textCursor()$selectedText(), "'")
})
```

The `copyAvailable` signal is largely equivalent, except it passes a boolean argument indicating whether the selection is non-empty.

Formatting By default, the widget will wrap text as entered. For use as a code editor, this is not desirable. The `lineWrapMode` takes values from the enumeration `QTextEdit::LineWrapMode` to control this:

```
te$lineWrapMode <- Qt$QTextEdit$NoWrap
```

The `setAlignment` method aligns the current paragraph (the one with the cursor) with values from `Qt::Alignment`.

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:

```
te <- Qt$QTextEdit(LoremIpsum)          # some text
te$find("qui", Qt$QTextDocument$FindWholeWords)
```

```
[1] TRUE
```

```
te$textCursor()$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 Section 15, one 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$QTextEdit)
#
qsetMethod("contextMenuEvent", QTextEditWithCompletions, function(e) {
  m <- this$createStandardContextMenu()
  if(this$textCursor()$hasSelection()) {
    selection <- this$textCursor()$selectedText()
    comps <- utils:::matchAvailableTopics(selection)
    comps <- setdiff(comps, selection)
    if(length(comps) > 0 && length(comps) < 10) {
      m$addSeparator()                  # add actions
      sapply(comps, function(i) {
        a <- Qt$QAction(i, this)
        qconnect(a, "triggered", function(checked) {
          insertPlainText(i)
        })
        m$addAction(a)
      })
    }
  }
  m$exec(e$globalPos())
})
te <- QTextEditWithCompletions()
```

The `createStandardContextMenu` method returns the base context menu, including functions like copy and paste. We add an action for every possible completion. Triggering an action will paste the completion into the document.

Example 14.6: A tabbed text editor

This example shows how to combine the text edit with a notebook widget to create a widget for editing more than one file at a time. We begin with a sub-class of `QTextEdit` that specially represents a editing component in a notebook of a text editor. The constructor initializes some parameters and connects handlers to update the application actions when the state of the editor changes:

```
qsetClass("TextEditSheet", Qt$QTextEdit, function(parent=NULL) {
  super(parent)
  this$lineWrapMode <- Qt$QTextEdit$NoWrap
  setFontFamily("Courier")

  actions <- window()$actions()
```

```
  qconnect(this, "redoAvailable", function(yes) {
    if(isCurrentSheet())
      actions$redo$setEnabled(yes)
  })
  qconnect(this, "undoAvailable", function(yes) {
    if(isCurrentSheet())
      actions$undo$setEnabled(yes)
  })
  qconnect(this, "selectionChanged", function() {
    hasSelection <- this$textCursor()$hasSelection()
    if(isCurrentSheet()) {
      actions$cut$setEnabled(hasSelection)
      actions$copy$setEnabled(hasSelection)
    }
  })
  qconnect(this, "textChanged", function() {
    if(isCurrentSheet()) {
      mod <- this$document()$isModified()
      actions$save$setEnabled(mod)
    }
  })
})
```

In this simple example, the `TextEditSheet` will keep track of its file name. In a real design, the filename would probably be associated with the underlying data. The file name is used for saving the sheet and for labeling the notebook tab. The latter requires special logic in the setter for the property:

```
qsetProperty("filename", TextEditSheet, write = function(fname) {
  this$filename <- fname
  ## update tab label
  notebook <- this$window()$notebook()
  ind <- notebook$indexOf(this)
  notebook$setTabText(ind, basename(fname))
})
```

Next, we define a few methods for the sheet. First, one to save the file. We use the filename property as the default destination.

```
qsetMethod("saveSheet", TextEditSheet, function(fname = filename) {
  txt <- this$toPlainText()
  writeLines(strsplit(txt, "\n")[[1]], con=fname)
})
```

The next method returns whether this editing sheet is current one in the application:

```
qsetMethod("isCurrentSheet", TextEditSheet, function() {
  notebook <- this$window()$notebook()
```

```
  notebook$currentIndex == notebook$indexOf(this)
})
```

Next, we construct the window that contains the notebook of documents. We subclass `QMainWindow` (see Chapter 15), so that we can add a toolbar. Our constructor customizes the notebook, sets up the actions and toolbar, then opens with a blank sheet:

```
qsetClass("TextEditWindow", Qt$QMainWindow, function(parent=NULL) {
  super(parent)

  notebook <- Qt$QTabWidget()
  notebook$tabsClosable <- TRUE
  notebook$usesScrollButtons <- TRUE
  notebook$documentMode <- TRUE
  qconnect(notebook, "tabCloseRequested",
           function(ind) notebook$removeTab(ind))
  qconnect(this$notebook, "currentChanged", function(ind) {
    if(ind > 0)
      updateActions()
  })
  setCentralWidget(notebook)

  initActions()
  makeToolbar()
  newSheet()
})
```

We have several actions possible in our GUI, such as the standard cut, copy and paste. We define them for the entire application, but the actions primarily work at the sheet level. The `initActions` constructs the actions and adds them to the window:

```
qsetMethod("initActions", TextEditWindow, function() {
  makeSheetAction <- function(name) {
    x <- Qt$QAction(name, this)
    x$setShortcuts(Qt$QKeySequence[[name]])
    qconnect(x, "triggered", function() {
      get(tolower(x$text), currentSheet())()
    })
    addAction(x)
    x
  }

  sapply(c("Redo", "Undo", "Cut", "Copy", "Paste", "Save"), makeSheetAction)

  x <- Qt$QAction("open", this)
  x$setShortcuts(Qt$QKeySequence$Open)
  qconnect(x, "triggered", function() {
```

```
    fname <- Qt$QFileDialog$getOpenFileName(this, "Select a file...", getwd())
    openSheet(fname)
  })
  addAction(x)

  x <- Qt$QAction("new", this)
  x$setShortcuts(Qt$QKeySequence$New)
  qconnect(x, "triggered", newSheet)
  addAction(x)
})
```

The enclosed `makeSheetAction` function creates an action that calls a method of the same name on the current sheet.

The `makeToolbar` method adds the actions to a toolbar:

```
qsetMethod("makeToolbar", TextEditWindow, function() {
  tb <- Qt$QToolBar()
  a <- actions()
  lapply(a, tb$addAction)
  tb$insertSeparator(a$cut)
  tb$insertSeparator(a$undo)
  this$addToolBar(tb)
})
```

Our GUI might also benefit from a menu bar, an exercise left for the reader.

This method opens a new sheet, with optional initial text:

```
qsetMethod("newSheet", TextEditWindow,
function(title="*scratch*", str="") {
  a <- TextEditSheet()          # a new sheet

  this$notebook$addTab(a, "")  # add to the notebook
  ind <- this$notebook$indexOf(a)
  this$notebook$setCurrentIndex(ind)

  a$setPlainText(str)
  a$setFilename(title)          # also updates tab
})
```

The `openSheet` method reads the text of a given file and displays it in a new sheet:

```
qsetMethod("openSheet", TextEditWindow, function(fname) {
  txt <- paste(readLines(fname), collapse="\n")
  newSheet(fname, txt)
})
```

The `updateActions` method is called whenever the current sheet changes, so that the state of the actions reflect that sheet:

```
qsetMethod("updateActions", TextEditWindow, function() {
```

```
  cur <- currentSheet()
  if(is.null(cur))
    return()
  a <- actions()
  a$redo$enabled <- FALSE
  a$undo$enabled <- FALSE
  a$cut$enabled <- cur$textCursor()$hasSelection()
  a$copy$enabled <- cur$textCursor()$hasSelection()
  a$paste$enabled <- cur$canPaste()
})
```

Finally, we define some accessors for getting the tabbed notebook, the currently edited sheet and the list of application actions:

```
qsetMethod("notebook", TextEditWindow, function() {
  this$centralWidget()
})
qsetMethod("currentSheet", TextEditWindow, function() {
  this$notebook()$currentWidget()
})
qsetMethod("actions", TextEditWindow, function() {
  actions <- super("actions")
  names(actions) <- sapply(actions, `$`, "text")
  actions
})
```

Syntax highlighting The text edit widget supports syntax highlighting through the `QSyntaxHighlighter` class. To implement a specific highlighting rule, one 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.

15

Qt: Application Windows

Many applications have a central window that typically contains a menubar, toolbar, an application-specific area, and a statusbar 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 menubar, toolbar and status bar, and also provides a framework for dockable windows.

To demonstrate the `QMainWindow` framework, we will create a simple spreadsheet application. First, we construct a `QMainWindow` object:

```
mainWindow <- Qt$QMainWindow()
```

The region between the toolbar and statusbar, 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)
tableView <- Qt$QTableView()
tableView$setModel(model)
mainWindow$setCentralWidget(tableView)
```

We will continue by adding a menubar and toolbar to our window. This depends on an understanding of how Qt represents actions.

Actions

The buttons in the menubar 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:

```
openAction <- Qt$QAction("Open", mainWindow)
```

The label text is passed to the constructor. 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:

```
openAction$statusTip <- "Load a spreadsheet from a CSV file"
```

One could also set an icon from a file:

```
iconFile <- system.file("images/open.gif", package="gWidgets")
openAction$setIcon(Qt$QIcon(iconFile))
```

Actions emit a `triggered` signal when activated. The application should connect to this signal to implement the command behind the action:

```
qconnect(openAction, "triggered", function() {
  filename <- Qt$QFileDialog$getOpenFilename()
  tableView$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 that toggled whether the spreadsheet will be saved on exit:

```
saveOnExitAction <- Qt$QAction("Save on exit", mainWindow)
saveOnExitAction$checkable <- TRUE
```

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:

```
justGroup <- Qt$QActionGroup(mainWindow)
leftAction <- Qt$QAction("Left Align", justGroup)
leftAction$checkable <- TRUE
rightAction <- Qt$QAction("Right Align", justGroup)
rightAction$checkable <- TRUE
centerAction <- Qt$QAction("Center", justGroup)
centerAction$checkable <- TRUE
```

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:

```
openAction$setShortcut(Qt$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.

Menubars

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 menubar is the top-level entry point to the hierarchy. The placement of the menubar depends on the platform. On Mac OS X, applications share a menubar area at the top of the screen. On other platforms, the menubar is typically found at the top of the main window for the application.

We create an instance of `QMenuBar`:

```
menubar <- Qt$QMenuBar()
```

A `QMenuBar` is a container of `QMenu` objects, which represent the submenus. We create a `QMenu` for the "File" and "Edit" menus and add them to the menubar:

```
fileMenu <- Qt$QMenu("File")
menubar$addMenu(fileMenu)
editMenu <- Qt$QMenu("Edit")
menubar$addMenu(editMenu)
```

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:

```
fileMenu$addAction(openAction)
fileMenu$addSeparator()
fileMenu$addAction(saveOnExitAction)
fileMenu$addSeparator()
quitAction <- fileMenu$addAction("Quit")
justMenu <- editMenu$addMenu("Justification")
justMenu$addAction(leftAction)
justMenu$addAction(rightAction)
justMenu$addAction(centerAction)
```

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.

Context menus

Sometimes, actions pertain to a single widget or portion of a widget, instead of the entire application. In such cases, the menubar 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 simplest approach to providing a context menu involves two steps. First, add the desired actions to the widget:

```
sortMenu <- Qt$QMenu("Sort by")
sapply(colnames(qdataFrame(model)), sortMenu$addAction)
tableView$addAction(sortMenu$menuAction())
```

Second, we configure the widget to display a menu of the actions when a context menu is requested:

```
tableView$contextMenuPolicy <- Qt$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 popup window could list possible code completions. Under the default context menu policy, we can implement this logic in an override of the `contextMenuEvent` virtual method:

```
showCompletionPopup <- function(e) {
  popup <- Qt$QMenu()
  comps <- utils:::matchAvailableTopics(this$text)
  comps <- head(comps, 10) # trim if large
  sapply(comps, function(i) {
    a <- popup$addAction(i)
    qconnect(a, "triggered", function(...) this$setText(i))
  })
  popup$popup(e$globalPos())
}
qsetClass("CodeEntry", Qt$QLineEdit)
qsetMethod("contextMenuEvent", CodeEntry, showCompletionPopup)
e <- CodeEntry()
```

If subclassing is undesirable, one could change the context menu policy and connect to the signal `customContextMenuRequested`:

```
e <- Qt$QLineEdit()
```

```
e$contextMenuPolicy <- Qt$Qt$CustomContextMenu
qconnect(e, "customContextMenuRequested", showCompletionPopup)
```

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$QToolBar()
mainWindow$addToolBar(toolbar)
```

```
NULL
```

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

1. `addAction` to add an action,
2. `addWidget` to embed an arbitrary widget into the toolbar,
3. `addSeparator` to place a sepearator between items.

Before adding some actions to our toolbar, we define a function `getIcon` that loads a `QIcon` from a file in the `gWidgets` package:

We create each action, set its icon, and store it in a list for ease of manipulation at a later time in the program:

```
fileActions <- list()
fileActions$open <- Qt$QAction("Open", mainWindow)
fileActions$open$setIcon(getIcon("open"))
fileActions$save <- Qt$QAction("Save", mainWindow)
fileActions$save$setIcon(getIcon("save"))
plotActions <- list()
plotActions$barplot <- Qt$QAction("Barplot", mainWindow)
plotActions$barplot$setIcon(getIcon("barplot"))
plotActions$boxplot <- Qt$QAction("Boxplot", mainWindow)
plotActions$boxplot$setIcon(getIcon("boxplot"))
```

Finally, we add the actions to the toolbar, with a separator between the file actions and plot actions:

```
sapply(fileActions, toolbar$addAction)
toolbar$addSeparator()
sapply(plotActions, 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 `orientationroperty`.

Statusbars

Main windows reserve an area for a statusbar 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 statusbar is an instance of the `QStatusBar` class. We create one and add it to our window:

```
statusbar <- Qt$QStatusBar()
mainWindow$setStatusBar(statusbar)
```

There are three types of messages in a statusbar:

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, one can embed widgets into the statusbar.

We could communicate a temporary message when a dataset is loaded:

```
statusbar$showMessage("Load complete", 1000)
```

The second argument 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 statusbar like any other widget:

```
statusbar$addWidget(Qt$QLabel("Ready"))
statusbar$addPermanentWidget(Qt$QLabel("Version 1.0"))
```

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$QDockWidget()
dock$setWidget(device)
```

```
NULL
```

By default, the dock widget is closable, movable and floatable. This is adjustable through the `features`roperty. For example, we could disable closing of the graphics device:

```
dock$features <- Qt$QDockWidget$DockWidgetMovable |
                 Qt$QDockWidget$DockWidgetFloatable
```

The `allowedAreas`roperty 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:

```
mainWindow$addDockWidget(Qt$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$QDockWidget(device2)
mainWindow$tabifyDockWidget(dock, dock2)
```

To make `dock2` a top-level window instead, we could set the `floating`roperty to `"TRUE"`:

```
dock2$floating <- TRUE
```

Example 15.1: A main window for an IDE

This example shows how to begin constructing a main window (Figure **??**) similar to that of the web application R-Studio. (`rstudio.org`).

We begin by setting a minimum size and a title for the main window:

```
w <- Qt$QMainWindow()
w$setMinimumSize(800, 500)
w$setWindowTitle("Rstudio-type layout")
```

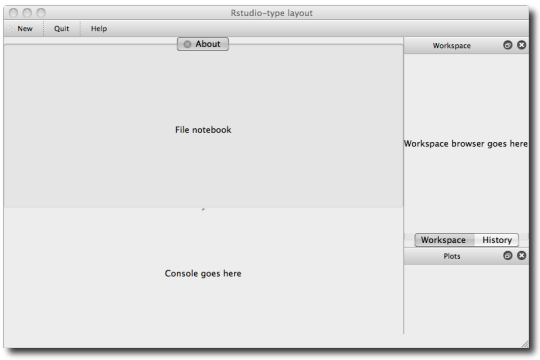

Figure 15.1: A GUI using dockable widgets and a `QMainWindow` instance

We add a menu bar and toolbar. Here only the file menu definitions are shown:

```
l <- list()
mb <- Qt$QMenuBar()
w$setMenuBar(mb)
fmenu <- mb$addMenu("File")
fmenu$addAction(l$new <- Qt$QAction("New", w))
fmenu$addSeparator()
fmenu$addAction(l$open <- Qt$QAction("Open", w))
fmenu$addAction(l$save <- Qt$QAction("Save", w))
fmenu$addSeparator()
fmenu$addAction(l$quit <- Qt$QAction("Quit", w))
```

The toolbar contains only some of the most important actions:

```
tb <- Qt$QToolBar()
w$addToolBar(tb)
tb$addAction(l$new)
tb$addSeparator()
tb$addAction(l$quit)
tb$addSeparator()
tb$addAction(l$help <- Qt$QAction("Help", w))
```

Our central widget holds two main areas: one for editing files and one for a console. A `QSplitter` divides the space between the two main widgets:

```
centralWidget <- Qt$QSplitter()
centralWidget$setOrientation(Qt$Qt$Vertical)
w$setCentralWidget(centralWidget)
```

As we may want to edit multiple files, we embed the editor widgets in a tabbed notebook, with closable tabs:

```
fileNotebook <- Qt$QTabWidget()
l <- Qt$QLabel("File notebook")
l$setAlignment(Qt$Qt$AlignCenter)
fileNotebook$addTab(l, "About")
fileNotebook$setTabsClosable(TRUE)
qconnect(fileNotebook, "tabCloseRequested", function(ind, nb) {
  nb$removeTab(ind)
}, user.data=fileNotebook)
centralWidget$addWidget(fileNotebook)
```

Our console widget is just a stub:

```
consoleWidget <- Qt$QLabel("Console goes here")
consoleWidget$setAlignment(Qt$Qt$AlignCenter)
centralWidget$addWidget(consoleWidget)
```

The right side of the layout will contain various tools for interacting with the R session. We place these into dock widgets, in case the user would like to place them elsewhere on the screen. We show a stub for a workspace browser:

```
workspaceBrowser <- Qt$QLabel("Workspace browser goes here")
wbDockWidget <- Qt$QDockWidget("Workspace")
wbDockWidget$setWidget(workspaceBrowser)
```

The workspace and history browser are placed in a notebook to conserve space. We add the workspace browser on the right side, then tabify the history browser (whose construction is not shown):

```
w$addDockWidget(Qt$Qt$RightDockWidgetArea, wbDockWidget)
w$tabifyDockWidget(wbDockWidget, hbDockWidget)
```

We next place a plot notebook in its own space:

```
plotNotebook <- Qt$QTabWidget()
pnDockWidget <- Qt$QDockWidget("Plots")
w$addDockWidget(Qt$Qt$RightDockWidgetArea, pnDockWidget)
```

Finally, we add a status bar and show a transient message:

```
sb <- Qt$QStatusBar()
w$setStatusBar(sb)
sb$showMessage("Mock-up layout for an IDE", 2000)
```

Part IV

The `tcltk` package

16

Tcl/Tk: Overview

Tcl ("tool command language") is a scripting language and interpreter of that language. Originally developed in the late 80s by John Ousterhout as a "glue" to combine two or more complicated applications together, it evolved overtime to find use not just as middleware, but also as a standalone development tool.

Tk [1] is an extension of Tcl that provides GUI components through Tcl. This was first developed in 1990, again by John Ousterhout. Tk quickly found widespread usage, as it made programming GUIs for X11 easier and faster. Over the years, other graphical toolkits have evolved and surpassed this one, but Tk still has numerous users.

Tk has a large number of bindings available for it, e.g. `Perl`, `Python`, `Ruby`, and through the `tcltk` package, R. The `tcltk` package was developed by Peter Dalgaard and included in R from version 1.1.0. Since then, the package has been used in a number of GUI projects for R, most notably, the `Rcmdr` GUI. In addition, 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] Tk has a well documented API (Tcl, a) (`www.tcl.tk/man/tcl8.5`). There are also several books to supplement. We consulted the one by Welch, Jones and Hobbs (Brent B. Welch, 2003) often in the developement of this material. In addition, the Tk Tutorial of Mark Roseman (Tcl, b) (`www.tkdocs.com/tutorial`) provides much detail. R specific documentation include two excellent R News articles and a proceedings paper (Dalgaard, 2001), (**?**) and (Dalgaard) by Peter Dalgaard, the package author. A set of examples due to James Wettenhall (Wettenhall) are also quite instructive. A main use of `tcltk` is within the `Rcmdr` framework. Writing extensions for that is well documented in an R News article (Fox, 2007) by John Fox, the package author.

Despite its limitations as a graphical toolkit, as compared to GTK+ or Qt, the Tk libraries are widely used for R GUIs, as for most users there are no installation issues. As of version 2.7.0, R for Windows has been bundled with the necessary Tk version, 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.

16.1 Interacting with Tcl

Although both are scripting languages, the basic syntax of Tcl is a bit unlike R. For example a simple string assignment would be made at tclsh, the Tcl shell with (using % as a prompt):

```
% set x {hello world}
hello world
```

Unlike R where braces are used to form blocks, this example shows how Tcl uses braces instead of quotes to group the words as a single string. The use of braces, instead of quotes, in this example is optional, but in general isn't, as expressions within braces are not evaluated.

The example above assigns to the variable `x` the value of `hello world`. Once assignment has been made, one can call commands on the value stored in `x` using the `$` prefix:

```
% puts $x
hello world
```

The `puts` command, in this usage, simply writes back its argument to the terminal. Had we used braces the argument would not have been substituted:

```
% puts {$x}
$x
```

More typical within the `tcltk` package is the idea of a subcommand. For example, the `string` command provides the subcommand `length` to return the number of characters in the string.

```
% string length $x
11
```

The `tcltk` package provides the low-level function `.Tcl` for direct access to the Tcl interpreter:

```
library(tcltk)
.Tcl("set x {some text}")                # assignment
```

```
<Tcl> some text

 .Tcl("puts $x")                          # 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 allowing arbitrary Tcl scripts to run within an R session. Tcl packages may be read in with `tclRequire`.

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. They differ in how they treat spaces and new lines. Conversion to numeric values is also possible through `as.numeric`, but conversion to logical may require two steps as only modes `character`, `double` or `integer` are stored. In general though, conversion of complicated Tcl expressions is not supported.

To simplify coercion to logical, we define a new method:

```
as.logical.tclObj <- function(x, ...) as.logical(as.numeric(x))
```

The Tk extensions to Tcl have a complicated command structure, and thankfully, `tcltk` provides some more conveniently named functions. To illustrate, the Tcl command to set the text value for a label object (`.label`) would look like

```
% .label configure -text "new text"
```

The `tcltk` package provides a corresponding function `tkconfigure`. The above would be done in an R-like way as (assuming `l` is a label object):

```
tkconfigure(l, 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 `l` above, in `l$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, one can specify nothing, returning a list of all options; just an option, to query the configured value; the option with a value, to modify the option; and possibly do more than one at at time. For

```
tcl(widget, subcommand, key=value, callback)
  /          |          |          \
widget$ID  subcommand   -key value   makeCallback
```

Figure 16.1: How the `tcl` function maps its arguments

commands such as `configure`, there will usually correspond a function in R of the same name with a `tk` prefix, as in this case `tkconfigure`. [2]

To make consulting the Tk manual pages easier in the text we would describe the configure subcommand as *ttklabel* `configure` *[options]* . (The R manual pages simply redirect you to the original Tk documentation, so understanding this is important for reading the API.) However, if such a function shortcut is present, we will typically use the shortcut when we illustrate code.

Some subcommands have further subcommands. An example is to set the selection. In the R function, the second command is appended with a dot, as in `tkselection.set`. (There are 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). It converts R `key=value` pairs into `-key value` options for Tcl. As named arguments are only for the `-key value` expansion, we follow the Tcl language and call the arguments "options" in the following. Finally, it adjusts any callback functions. The `tcl` function uses position to create its command. The order of the subcommands needs to match that of the Tk API, so although it is true that often the R object is first, this is not always the case.

16.2 Constructors

In this Chapter, we will stick to a few basic widgets: labels and buttons; and top-level containers to illustrate the basic usage of `tcltk`, leaving for later more detail on containers and widgets.

Unlike GTK+, say, the construction of widgets in `tcltk` is linked to the widget hierarchy. Tk widgets are constructed as children of a parent container with the parent specified to the constructor. When the Tk

[2] The package `tcltk` was written before namespaces were implemented in R, so the "tk" prefix serves that role.

shell, wish, is used or the Tk package is loaded through the Tcl command `package require Tk`, a top level window named "." is created. In the variable name `.label`, from above, the dot refers to the top level window. In `tcltk` a top-level window is created separately through the `tktoplevel` constructor, as with

```
w <- tktoplevel()
```

Top-level windows will be explained in more detail in Chapter 17. For now, we just use one to be a parent container for a label widget. Like all constructors but the one for toplevel windows, the label constructor (`ttklabel`) requires a specification of the parent container followed by any other options that are desired. A typical invocation would look like:

```
l <- ttklabel(w, text="label text")
```

To see the label, we can pack it into the toplevel window, which was used as its parent at construction.

```
tkpack(l)
```

Options The first argument of a widget constructor is the parent container, subsequent arguments are used to specify the options for the constructor given as `key=value` pairs. The Tk API lists these options along with their description.

For a simple label, the following options are possible: `anchor`, `background`, `font`, `foreground`, `justify`, `padding`, `relief`, `text`, and `wraplength`. This is in addition to the standard options `class`, `compound`, `cursor`, `image`, `state`, `style`, `takefocus`, `text`, `textvariable`, `underline`, and `width`. (Although clearly lengthy, this list is significantly reduced from the options for `tklabel` where options for the many style properties are also included.)

Many of the options are clear from their name. The main option, `text`, takes a character string. The label will be multiline if this 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 `c(left, top, right, bottom)`, or fewer, with `bottom` defaulting to `top`, `right` to `left` and `top` to `left`. The `relief` argument specifies how a 3-d effect around the label should look, if specified. Possible values are `"flat"`, `"groove"`, `"raised"`, `"ridge"`, `"solid"`, or `"sunken"`.

The functions tkconfigure, tkcget Option values may be set through the constructor, or adjusted afterwards by `tkconfigure`. A listing (in Tcl code) of possible options that can be adjusted may be seen by calling `tkconfigure` with just the widget as an argument.

```
head(as.character(tkconfigure(l)))      # first 6 only
```

```
[1] "-background frameColor FrameColor {} {}"
[2] "-foreground textColor TextColor {} {}"
[3] "-font font Font {} {}"
[4] "-borderwidth borderWidth BorderWidth {} {}"
[5] "-relief relief Relief {} {}"
[6] "-anchor anchor Anchor {} {}"
```

The tkcget function returns the value of an option (again as a tclObj object). The option can be specified two different ways. Either using the Tk style of a leading dash or using the R convention that NULL values mean to return the value, and not set it.

```
tkcget(l, "-text")                          # retrieve text property
```

```
<Tcl> label text
```

```
tkcget(l, text=NULL)                        # alternate syntax
```

```
<Tcl> label text
```

Coercion to character As mentioned, the tclObj objects can be coerced to character class two ways. The conversion through as.character breaks the return value along whitespace:

```
as.character(tkcget(l, text=NULL))
```

```
[1] "label" "text"
```

Whereas, conversion by the tclvalue function does not:

```
tclvalue(tkcget(l, text=NULL))
```

```
[1] "label text"
```

Buttons Buttons are constructed using the ttkbutton constructor.

```
b <- ttkbutton(w, text="click me",
               command=function() print("thanks"))
```

Buttons and labels share many of the same options. However, in addition, buttons have a command option to specify a callback for when the button is invoked. Buttons may be 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 *ttkbutton* invoke subcommand. In the above example, clicking on the button will call the function causing a simple message to be printed. More on callbacks in tcltk will be explained in Section 16.3.

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)

```
str(b)
```

```
List of 2
 $ ID : chr ".2.2"
 $ env:<environment: 0x100da8b50>
 - 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 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 (Dalgaard). As the construction of a new widget requires the `ID` and environment of its parent, the first argument to `tkwidget`, `parent`, must be an R Tk object, not simply its character ID, as is possible for the `tcl` function.

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(b, "instate", "focus"))
```

```
[1] FALSE
```

To set a widget to be not sensitive to user input we have:

```
tcl(b, "state", "disabled")              # not sensitive
```

```
<Tcl> !disabled
```

The states are bits and can be negated by prefacing the value with `!`:

```
tcl(b, "state", "!disabled")             # sensitive again
```

```
<Tcl> disabled
```

The full list of states is in the manual page for `ttk::widget`.

Geometry managers

As with Qt, when a new widget is constructed it is not automatically mapped. Tk uses geometry managers to specify how the widget will be drawn within the parent container. We will discuss two such geometry managers in Chapter 17, but for now, we note that the simplest way to view a widget in its parent window is through `tkpack`, as in:

```
tkpack(b)
```

This command packs the widgets into the top-level window (the parent in this case) in a box-like manner. Unlike GTK+ more than one child can be packed into a top-level window, although we don't demonstrate this further, as later we will use an intermediate `ttkframe` box container so that themes are properly displayed.

Tcl variables

For the button and label widgets, there is an option `textvariable` to specify a Tcl variable to store the text property. These variables are dynamically bound to the widget, so that changes to the variable are propagated to the GUI. (The Tcl variable is a model and the widget a view of the model.) Other widgets allow for tcl variables to be used for this and other purposes The basic functions involved are `tclVar` to create a Tcl variable, `tclvalue` to get the assigned value and `tclvalue<-` to modify the value.

```
textvar <- tclVar("another label")
l2 <- ttklabel(w, textvariable=textvar)
tkpack(l2)
tclvalue(textvar)
```

```
[1] "another label"
```

```
tclvalue(textvar) <- "new text"
```

Tcl variables have a unique identifier, returned by `as.character`:

```
as.character(textvar)
```

```
[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, their usage is recommended. One disadvantage, is that in a callback, the variable is not passed to the callback and must be found through R's scoping rules. (In Section 18.2 we show a workaround.)

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 only referred to 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
```

Window properties and state: `tkwinfo`

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. Finally, there is also `tkwinfo` to return the properties of the containing window of the widget. When widgets are mapped, the "window" they are rendered to has properties, such as a class or size. If the API is of the form

`winfo` **subcommand_name** *window*

the specification to `tkwinfo` is in the same order (the widget is not the first argument). For instance, the class [3] of a label is returned by the `class` subcommand:

```
tkwinfo("class", l)
```

```
<Tcl> TLabel
```

The window, in this example `l`, can be specified as an R object, or by its character ID. This is useful, as the return value of some functions is the ID. For instance, the `children` subcommand returns IDs. Below the `as.character` function will coerce these into a vector of IDs.

```
(children <- tkwinfo("children", w))
```

```
<Tcl> .2.1 .2.2 .2.3
```

```
sapply(as.character(children), function(i) tkwinfo("class", i))
```

[3] The class of a widget is more like a attribute and should not be confused with class in the object oriented sense. The class is used internally for bindings and styles.

```
$‘.2.1‘
<Tcl> TLabel

$‘.2.2‘
<Tcl> TButton

$‘.2.3‘
<Tcl> TLabel
```

There are several possible subcommands, here we list a few. The *tkwinfo* `geometry` sub command returns the location and size of the widgets' window in the form `width x height + x + y`; the sub commands *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` sub command is similar, only it checks that all parent windows are also mapped. For traversing the widget hierarchy, one has available the *tkwinfo* `parent` sub command 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.

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 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.) There is no built in command to return the theme, so we use `.Tcl` to call the appropriate Tcl command. The `names` sub command will return the available themes:

```
.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")
```

The writing of themes will not be covered, but in Example 17.4 we show how to create a new style for a button. This topic is covered in some detail in the Tk tutorial by Roseman.

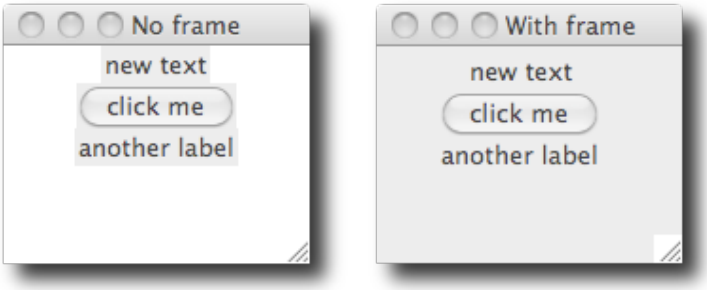

Figure 16.2: Similar GUIs, one using a frame within the toplevel window (right one) and one without. The one without has widgets whose background does not match the toplevel window.

The example we have shown so far, would not look quite right for some operating systems, as the toplevel window is not a themed widget (Figure 16.2). To work around that, a `ttkframe` widget is usually used to hold the child components of the top-level window. The following shows how to place a frame inside the window, with some arguments to be explained later that allow it to act reasonably if the window is resized.

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12))  # Some breathing room
tkpack(f, expand=TRUE, fill="both")     # For window expansion
l <- ttklabel(f,  text="label")         # some widget
tkpack(l)
```

Colors and fonts

Colors and fonts are typically specified through a theme, but at times it is desirable to customize the preset ones.

The label color can be set through its `foreground` property. Colors can be specified by name – for common colors – or by hex RGB values which are common in web programming.

```
tkconfigure(l, foreground="red")
tkconfigure(l, foreground="#00aa00")
```

To find the hex RGB value, one can use the `rgb` function to create RGB values from intensities in $[0,1]$. The R function `col2rgb` can translate a named color into RGB values. The `as.hexmode` function will display an integer in hexadecimal notation.

In Example 18.3 we show how to modify a style, as opposed to the `foreground` option, to change the text color in an entry widget.

Table 16.1: Standard font names defined by a theme.

Standard font name	Description
`TkDefaultFont`	Default font for all GUI items not otherwise specified
`TkTextFont`	Font for text widgets
`TkFixedFont`	Fixed-width font
`TkMenuFont`	Menu bar fonts
`TkHeadingFont`	Font for column headings
`TkCaptionFont`	Caption font (dialogs)
`TkSmallCaptionFont`	Smaller caption font
`TkIconFont`	Icon and text font

Fonts Fonts are a bit more involved than colors. Tk version 8.5 made it more difficult to change font properties of individual widgets, this following the practice of centralizing style options for consistency, ease of maintaining code and ease of theming. To set a font for a label, rather than specifying the font properties, one configures the `font` options using a pre-defined font name, such as

```
tkconfigure(l, font="TkFixedFont")
```

The `"TkFixedFont"` value is one of the standard font names, in this case to use a fixed-width font. A complete list of the standard names is provided in Table 16.2. Each theme sets these defaults accordingly.

tkfont.create The `tkfont.create` function can be used to create a new font, as with the following commands:

```
tkfont.create("ourFont", family="Helvetica", size=12,
              weight="bold")
```

```
<Tcl> ourFont
```

```
tkconfigure(l, font="ourFont")
```

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 16.3 shows a display of some available font families on a Mac OS X machine. See Example 18.9 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.

Figure 16.3: A scrollable frame widget (cf. Example 18.9) showing the available fonts on a system.

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` functions as follows:

```
chars <- c(0:9,LETTERS, letters)
tmp <- tkfont.measure("TkTextFont",paste(chars,collapse=""))
fontWidth <- ceiling(as.numeric(tclvalue(tmp))/length(chars))
tmp <- tkfont.metrics("TkTextFont","linespace"=NULL)
fontHeight <- as.numeric(tclvalue(tmp))
c(width=fontWidth, height=fontHeight)
```

```
width height
    8     15
```

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. 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.) 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.) One 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:

```
l <- ttklabel(w, image="::img::tclLogo", text="logo text",
              compound = "top")
```

By default the text will not show. The `compound` argument takes a value of either `"text"`, `"image"` (default), `"center"`, `"top"`, `"left"`, `"bottom"`, or `"right"` specifying where the label is in relation to the text.

Image manipulation Once an image is created, there are several options to manipulate the image. These are found in the Tk man page for `photo`, not `image`. For instance, to change the palette so that instead of `fullcolor` only 16 shades of gray are used to display the icon, one could issue the command

```
tkconfigure("::img::tclLogo", palette=16)
```

16.3 Events and Callbacks

The button widget has the `command` option for assigning a callback which is invoked (among other ways) when the user clicks the mouse on the button. In addition to such commands, one may use `tkbind` to invoke callbacks in response to many other events that the user may initiate.

The basic call is `tkbind(tag, event, script)`.

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 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.

```
w <- tktoplevel()
l <- ttkbutton(w, text="Some widget with the focus"); tkpack(l)
tkbind(w, "<Control-q>", function() tkdestroy(w))
```

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

The possible events (or sequences of events) vary from widget to widget. The 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`.

```
w <- tktoplevel()
l <- ttklabel(w, text="Click Ok for a message")
b1 <- ttkbutton(w, text="Cancel", command=function() tkdestroy(w))
b2 <- ttkbutton(w, text="Ok", command=function() {
  print("initiate an action")
})
sapply(list(l,b1,b2), tkpack)
tkbind(w, "C", function() tcl(b1, "invoke"))
tkconfigure(b1, underline=0)
#
tkbind(w, "O", function() tcl(b1, "invoke"))
tkconfigure(b2, underline=0)
tkfocus(b2)
#
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. 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> also has the detail `1`. The full list of modifiers and types are described in the man page for `bind`. Some familiar modifiers are `Control`, `Alt`, `Button1` (also `B1`), `Double` and `Triple`. The event types are the standard X event types along with some abbreviations. These are also listed in the `bind` man page. Some commonly used ones are `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 with <<EventName>>. There are predefined virtual events listed in the `event` man page. These include <<MenuSelect>> when working with menus, <<Modified>> for text widgets, <<Selection>> for text widgets, and <<Cut>>, <<Copy>> and <<Paste>> for working with the clipboard. New virtual events can be produced with the `tkevent.add` function. This takes at least two arguments, an event name and a sequence that will initiate that event. The `event` man page has these examples coming from the Emacs world:

```
tkevent.add("<<Paste>>", "<Control-y>")
tkevent.add("<<Save>>", "<Control-x><Control-s>")
```

In addition to virtual events occurring when the sequence is performed, the `tkevent.generate` can be used to force an event for a widget. This function requires a widget (or its ID) and the event name. Other options can be used to specify substitution values, described below. To illustrate, this command will generate the <<Save>> event for the button `b`:

```
tkevent.generate(b, "<<Save>>")
```

Callbacks

The `tcltk` package implements callbacks in a manner different from Tk, as the callback functions are R functions, not Tk procedures. This is much more convenient, but introduces some slight differences. In `tcltk` these callbacks can be expressions (unevaluated calls) or functions. We use only the latter, for more clarity. The basic callback function need not have any arguments and those that do only have 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 18.3) it can matter. [4]

In `tcltk` only one callback can be associated with a widget and event through the call `tkbind(widget,event,callback)`. (Although, callbacks for the widget associated with classes or toplevel windows can differ.) Calling `tkbind` another time will replace the callback. To remove a callback, simply assign a new callback which does nothing. [5]

% Substitutions

One can not 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` man 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, whereas Example 16.1 will put these to use.

```
w <- tktoplevel()
b <- ttkbutton(w, text="Click me to record the x,y position")
tkpack(b)
tkbind(b, "<ButtonPress-1>", function(W, x,y, X, Y) {
  print(W)                              # an ID
  print(c(x, X))                        # character class
```

[4] The 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 call backs does not work.

[5] This event handling can prevent being able to port some Tk code into `tcltk`. In those cases, one may consider sourcing in Tcl code directly.

```
  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:

```
   ID <- tcl("after", 1000, function() print("1 second passed"))
```

The ID returned by `after` may be used to `cancel` the command before it executes. To execute a command repeatedly, can be done along the lines of:

```
afterID <- ""
someFlag <- TRUE
repeatCall <- function(ms=100, f) {
  afterID <<- tcl("after", ms, function() {
    if(someFlag) {
      f()
      afterID <<- repeatCall(ms, f)
    }  else {
      tcl("after", "cancel", afterID)
    }
  })
}
repeatCall(2000, function() {
  print("Running. Set someFlag <- FALSE to stop.")
})
```

Example 16.1: Drag and Drop
This relatively involved example [6] shows several different uses of the event framework to implement drag and drop behavior between two widgets. In `tcltk` much more work is involved with drag and drop, than with `RGtk2` and `qtbase`. Steps are needed to make one widget a drop source, and other steps are needed to make the other widget a drop target.

The basic idea is that when a value is being dragged, virtual events are generated for the widget the cursor is over. If that widget has callbacks bound to these events, then the drag and drop can be processed.

To begin, we create a simple GUI to hold three widgets. We use buttons for drag and drop, but only because we haven't yet discussed more natural widgets such as the text widgets.

```
w <- tktoplevel()
bDrag <- ttkbutton(w, text="Drag me")
```

[6] The idea for the example code originated with `http://wiki.tcl.tk/416`

```
bDrop <- ttkbutton(w, text="Drop here")
tkpack(bDrag)
tkpack(ttklabel(w, text="Drag over me"))
tkpack(bDrop)
```

Before beginning, we define three global variables that can be shared among drop sources to keep track of the drag and drop state.

```
.dragging <- FALSE                  # currently dragging?
.dragValue <- ""                    # value to transfer
.lastWidgetID <- ""                 # 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.

```
tkbind(bDrag,"<ButtonPress-1>",function(W) {
  .dragging <<-  TRUE
  .dragValue <<- as.character(tkcget(W,text=NULL))
  .lastWidgetID <<- 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 one 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 over. We use `tkwinfo` to find the widget containing the x,y-coordinates of the cursor position. We then generate the `<<DragOver>>` virtual event for this widget, and if this widget is different from the previous last widget, we generate the `<<DragLeave>>` virtual event.

```
tkbind(w, "<B1-Motion>", function(W, X, Y) {
  if(!.dragging) return()
  ## see cursor help page in API for more options
  ## For custom cursors cf. http://wiki.tcl.tk/8674.
  tkconfigure(W, cursor="coffee_mug")   # set cursor

  w = tkwinfo("containing", X, Y)       # widget mouse is over
  if(as.logical(tkwinfo("exists", w)))  # does widget exist?
    tkevent.generate(w, "<<DragOver>>")

  ## generate drag leave if we left last widget
  if(as.logical(tkwinfo("exists", w)) &&
     nchar(as.character(w)) > 0 &&
     length(.lastWidgetID) > 0          # if not tcltk character(0)
```

```
      ) {
     if(as.character(w) != .lastWidgetID)
       tkevent.generate(.lastWidgetID, "<<DragLeave>>")
   }
   .lastWidgetID <<- as.character(w)
 })
```

Finally, if the button is released, we generate the <<DragLeave>> and, most importantly, <<DragDrop>> virtual events for the widget we are over.

```
 tkbind(bDrag, "<ButtonRelease-1>", function(W, X, Y) {
  if(!.dragging) return()
  w <- tkwinfo("containing", X, Y)

  if(as.logical(tkwinfo("exists", w))) {
    tkevent.generate(w, "<<DragLeave>>")
    tkevent.generate(w, "<<DragDrop>>")
    tkconfigure(w, cursor="")
  }
  .dragging <<- FALSE
  .lastWidgetID <<- ""
  tkconfigure(W, cursor="")
 })
```

To set up a drop target, we bind callbacks for the virtual events generated above to the widget. For the <<DragOver>> event we make the widget active so that it appears ready to receive a drag value.

```
tkbind(bDrop,"<<DragOver>>",function(W) {
  if(.dragging)
    tcl(W, "state", "active")
})
```

If the drag event leaves the widget without dropping, we change the state back to not active.

```
tkbind(bDrop,"<<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 .dragValue.

```
tkbind(bDrop,"<<DragDrop>>", function(W) {
  tkconfigure(W, text=.dragValue)
  .dragValue <- ""
})
```

17

Tcl/Tk: Layout and Containers

17.1 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 menubar through the `menu` argument. (Menus will be covered in Section 18.5.)

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. As with all such functions, either the top-level window object, or its ID must be the first argument. In this case, the new title is the second.

Suppressing the initial drawing When a top-level window is constructed there is no option for it not to be shown. However, one can use the `tclServiceMode` function to suspend/resume drawing of any widget through Tk. This function takes a logical value indicating the updating of widgets should be suspended. One can set the value to `FALSE`, initiate the widgets, then set to `TRUE` to display the widgets. 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 means 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 `tkwinfo` in Section 16.2. If this string is empty, then the window will resize to accomodate 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 toplevel window `w`.

```
tkwm.resizable(w, FALSE, FALSE)    # width first
```

When a window is resized, you can constrain the minimum and maximum sizes with `tkwm.minsize` and `tkwm.maxsize`. The aspect ratio (width/height) can be set through `tkwm.aspect`.

For 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 their mouse for resizing the window. On some OSes (e.g., Mac OS X) these are added by the window manager automatically.

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 17.1). The new window will mirror the state of the master window, including if the master is withdrawn.

For some dialogs it may be desirable to not have the window manager decorate the window with a title bar etc. The command *tktoplevel* `wm overrideredirect` *logical* takes a logical value indicating if the window should be decorated. Though, not all window managers respect this.

Bindings Bindings for top-level windows are propagated down to all of their child widgets. If a common binding is desired for all the children, then it need only be specified once for the top-level window (cf. Section 16.3 where keyboard accelerators 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 windows decorations. A top-level window can be removed through the `tkdestroy` function, or through the user clicking on the correct window decorations. When the window decoration is clicked, the window manager issues a `"WM_DELETE_WINDOW"` event. To bind to this, a command of this form `tkwm.protocol(win, "WM_DELETE_WINDOW", callback)` is used.

To illustrate, if `w` is a top-level window, and `e` a text entry widget (cf. `tktext` in Section 18.3) then the following snippet of code would check to see if the text widget has been modified before closing the window. This uses a modal message box described in Section 18.1.

```
tkwm.protocol(w,"WM_DELETE_WINDOW", function() {
  modified <- tcl(e, "edit", "modified")
  if(as.logical(modified)) {
```

```
    response <-
      tkmessageBox(icon="question",
                   message="Really close?",
                   detail="Changes need to be saved",
                   type="yesno",
                   parent=w)
    if(as.character(response) == "no")
      return()
  }
  tkdestroy(w)                          # otherwise close
})
```

Example 17.1: 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) {
  w <- tktoplevel()

  if(!missing(title)) tkwm.title(w, title)

  if(!missing(command))
    tkwm.protocol(w, "WM_DELETE_WINDOW", function() {
      if(command())            # command returns logical
        tkdestroy(w)
    })

  if(!missing(parent)) {
    parentWin <- tkwinfo("toplevel", parent)
    if(as.logical(tkwinfo("viewable", parentWin))) {
      tkwm.transient(w, parent)
    }
  }

  if(!missing(width)) tkconfigure(w, width=width)
  if(!missing(height)) tkconfigure(w, height=height)

  windowSystem <- tclvalue(tcl("tk", "windowingsystem"))
  if(windowSystem == "aqua") {
    f <- ttkframe(w, padding=c(3,3,12,12))
  } else {
    f1 <- ttkframe(w, padding=0)
    tkpack(f1, expand=TRUE, fill="both")
    f <- ttkframe(f1, padding=c(3,3,12,0))
    sg <- ttksizegrip(f1)
```

```
    tkpack(sg, side="bottom", anchor="se")
  }
  tkpack(f, expand=TRUE, fill="both", side="top")

  return(f)
}
```

17.2 Frames

The `ttkframe` constructor produces a themeable container that can be used to organize visible components within a GUI. 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 to put space within the border 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"`, or `"sunken"`.

Label Frames

The `ttklabelframe` constructor produces a frame with an optional label that can be used to set off and organize components of a GUI. The label is set through the option `text`. Its position is determined by the option `labelanchor` taking values labeled by compass headings (combinations of `n`, `e`, `w`, `s`. The default is theme dependent, although typically `"nw"` (upper left).

Separators As an alternative to a border, the `ttkseparator` widget can be used to place a single line to separate off 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.

17.3 Geometry Managers

Tcl uses *geometry managers* to place child components within their parent windows. There are three such managers, but we describe only two, leaving the lower-level `place` command for the official documentation. The use of geometry managers, allows Tk to quickly reallocate space to a GUI's components when a window is resized. The `tkpack` function will place children into their parent in a box-like manner. We have seen several examples in the text that use nested boxes to construct quite flexible layouts.

Example 17.3 will illustrate that once again. When simultaneous horizontal and vertical alignment of child components is desired, the `tkgrid` function can be used to manage the components.

A GUI may use a mix of `pack` and `grid` to mangage 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. The `side` option can take a value of `"left"`, `"right"`, `"top"` (default), or `"bottom"` to adjust where the children are placed. These can be mixed and matched, but sticking to just one direction is typical, with nested frames to give additional flexibility.

after, before The `after` and `before` options can be used to place the child before or after another component. These are used as with `tkpack(child1, after=child2)`. The object `child2` can be an R object or its ID.

forget Child components can be forgotten by the window manager, unmapping them but not destroying them, with the *tkpack* `forget` subcommand, or the convenience function `tkpack.forget`. After a child component is removed this way, it can be re-placed in the GUI using a geometry manager.

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 one might implement a ticker tape effect, where words scroll to the left.

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
#
x <- strsplit("Lorem ipsum dolor sit amet ...", "\\s")[[1]]
labels <- lapply(x, function(i) ttklabel(f, text=i))
sapply(labels, function(i) tkpack(i, side="left"))
#
```

Figure 17.1: 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).

```
rotateLabel <- function() {
  children <- as.character(tkpack("slaves", f))
  tkpack.forget(children[1])
  tkpack(children[1], after=children[length(children)],
         side="left")
}
```

One 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 17.1 has an example. In the above options, the `x` and `y` indicate left-right space or top-bottom space. The `i` stands for internal padding that is left on the sides or top and bottom of the child within the border, for `padx` the external padding added around the border of the child component. The value can be a single number or pair of numbers for asymmetric padding.

This sample code shows how one can easily add padding around all the children of the frame `f` using the *tkpack* `"configure"` subcommand.

```
allChildren <- as.character(tkwinfo("children", f))
sapply(allChildren, function(i) {
  tkpack("configure", i,  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

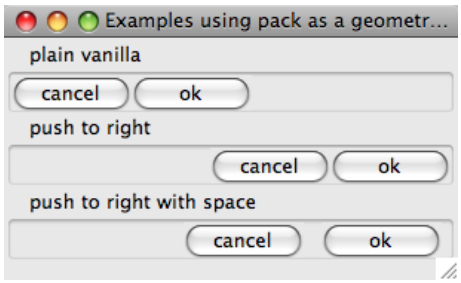

Figure 17.2: Demonstration of using `tkpack` options showing effects of using the `side` and `padx` options to create dialog buttons.

unallocated space called a "cavity." We use the nicer terms "child component" and "box" to describe these. When a child is placed inside the box, say on the top, the space allocated to the child is the rectangular space with width given by the width of the box, and height the sum of the requested height of the child plus twice the `ipady` amount (or the sum if specified with two numbers). The packer then chooses the dimension of the child component, again from the requested size plus the `ipad` values for `x` and `y`. These two spaces may, of course, have different dimensions.

By default, the child will be placed centered along the edge of the box within the allocated space and blank space, if any, on both sides. If there is not enough space for the child in the allocated space, the component can be drawn oddly. Enlarging the top-level window can adjust this.

anchor, expand, fill When there is more space in the box than requested by the child component, there are other options. The `anchor` option can be used to anchor the child to a place in the box by specifying one of the valid compass points (eg. `"n"` or `"se"`) leaving blank space around the child.

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 expanding the child's size in just one direction, the latter in both.

Example 17.2: Packing dialog buttons

This example shows how one can pack in action buttons, such as when a dialog is created.

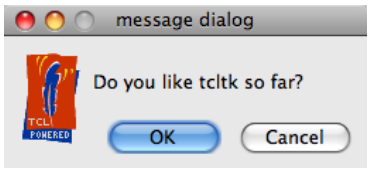

Figure 17.3: Example of a simple dialog

The first example just uses `tkpack` without any arguments except the side to indicate the buttons are packed in left to right, not top to bottom.

```
f1 <- ttklabelframe(f, text="plain vanilla")
tkpack(f1, expand=TRUE, fill="x")
l <- function(f)
  list(ttkbutton(f, text="cancel"), ttkbutton(f, text="ok"))
sapply(l(f1), function(i) tkpack(i, side="left"))
```

Typically the buttons are right justified. One way to do this is to pack in using `side` with a value of `"right"`. This shows how to use a blank expanding label to take up the space on the left.

```
f2 <- ttklabelframe(f, text="push to right")
tkpack(f2, expand=TRUE, fill="x")
tkpack(ttklabel(f2, text=" "), expand=TRUE, fill="x",
       side="left")
sapply(l(f2), function(i) tkpack(i, side="left"))
```

Finally, we add in some padding to conform to Apple's design specification that such buttons should have a 12 pixel separation.

```
f3 <- ttklabelframe(f, text="push to right with space")
tkpack(f3, expand=TRUE, fill="x")
tkpack(ttklabel(f3, text=" "), expand=TRUE, fill="x",
       side="left")
sapply(l(f3), function(i) tkpack(i, side="left", padx=6))
```

Example 17.3: A non-modal dialog
This example shows how to use a window, frames, labels, buttons, icons, packing and bindings to create a non-modal dialog.

Although not written as a function, we set aside the values that would be passed in were it.

```
title <- "message dialog"
message <- "Do you like tcltk so far?"
parent <- NULL
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.

```
w <- tktoplevel(); tkwm.title(w, title)
tkwm.state(w, "withdrawn")
f <- ttkframe(w,  padding=c(3, 3, 12, 12))
tkpack(f, expand=TRUE, fill="both")
```

As usual, we added a frame so that any themes are respected.

If the parent is non-null and is viewable, then the dialog is made transient to a parent, The parent need not be a top-level window, so `tkwinfo` if used to find the parent's top-level window. For Mac OS X, we use the `notify` attribute to bounce the dock icon until the mouse enters the window area.

```
if(!is.null(parent)) {
  parentWin <- tkwinfo("toplevel", parent)
  if(as.logical(tkwinfo("viewable", parentWin))) {
    tkwm.transient(w, parent)
    if(as.character(tcl("tk", "windowingsystem")) == "aqua") {
      tcl("wm","attributes",parentWin, notify=TRUE) # bounce
      tkbind(parentWin,"<Enter>", function()        # stop
             tcl("wm","attributes",parentWin, notify=FALSE))
    }
  }
}
```

We will use a standard layout for our dialog with an icon on the left, a message and buttons on the right. We pack the icon into the left side of the frame,

```
l <- ttklabel(f, image="::img::tclLogo", padding=5) # recycle
tkpack(l,side="left")
```

A nested frame will be used to layout the message area and button area. Since the `tkpack` default is to pack in top to bottom, no `side` specification is made.

```
f1 <- ttkframe(f)
tkpack(f1, expand=TRUE, fill="both")
#
m <- ttklabel(f1, text=message)
tkpack(m, expand=TRUE, fill="both")
```

The buttons have their own frame, as they are layed out horizontally.

```
f2 <- ttkframe(f1)
tkpack(f2)
```

The callback function for the OK button prints a message then destroys the window.

```
okCB <- function() {
  print("That's great")
  tkdestroy(w)
}
okButton <- ttkbutton(f2, text="OK", command=okCB)
cancelButton <- ttkbutton(f2, text="Cancel",
                          command=function() tkdestroy(w))
#
tkpack(okButton, side="left", padx=12)  # give some space
tkpack(cancelButton)
```

As our interactive behavior is consistent for both buttons, we make a binding to the `TButton` class, not 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(w, "normal")
tkwm.resizable(w, FALSE, FALSE)
tkfocus(okButton)
```

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, 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`

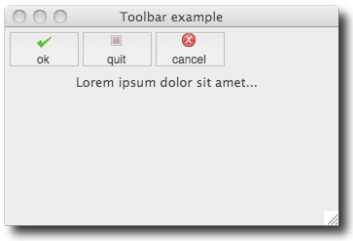

Figure 17.4: Illustration of using `tkpack` and `tkgrid` to make a toolbar.

functions through the option `weight`. The weight is a value between 0 and 1. If there are just two rows, and the first row has weight 1/2 and the second weight 1, then the extra space is allocated twice as much for the second row. The specific row or column must also be specified. Again. rows and columns are referenced starting with 0 not the 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` and `expand` and `fill` to control what happens when more space is available then 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 the direction like `expand=TRUE, fill="both"`.

Padding As with `tkpack`, `tkgrid` has options `ipadx`, `ipady`, `padx`, and `padx` 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 17.4: Using `tkgrid` to create a toolbar

Tk does not have a toolbar widget. Here we use `tkgrid` to show how we can add one to a top-level window in a manner that is not affected by resizing. We begin by packing a frame into a top-level window.

```
w <- tktoplevel(); tkwm.title(w, "Toolbar example")
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
```

Our example has two main containers: one to hold the toolbar buttons and one to hold the main content.

```
tbFrame <- ttkframe(f, padding=0)
contentFrame <- ttkframe(f, padding=4)
```

The `tkgrid` geometry manager is used to place the toolbar at the top, and the content frame below. The choice of sticky and the weights ensure that the toolbar does not resize if the window does.

```
tkgrid(tbFrame, row=0, column=0, sticky="we")
tkgrid(contentFrame, row=1, column=0, sticky = "news")
tkgrid.rowconfigure(f, 0, weight=0)
tkgrid.rowconfigure(f, 1, weight=1)
tkgrid.columnconfigure(f, 0, weight=1)
#
txt <- "Lorem ipsum dolor sit amet..." # sample text
tkpack(ttklabel(contentFrame, 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 `makeIcon` function finds stock icons from the `gWidgets` package and adds them to a button.

```
makeIcon <- function(parent, stockName, command=NULL) {
  iconFile <- system.file("images",
                          paste(stockName,"gif",sep="."),
                          package="gWidgets")
  if(nchar(iconFile) == 0) {
    b <- ttkbutton(parent, text=stockName, width=6)
  } else {
    iconName <- paste("::img::",stockName, sep="")
    tkimage.create("photo", iconName, file = iconFile)
    b <- ttkbutton(parent, image=iconName,
                   style="Toolbar.TButton", text=stockName,
                   compound="top", width=6)
    if(!is.null(command))
```

```
      tkconfigure(b, command=command)
  }
  return(b)
}
```

To illustrate, we pack in some icons. Here we use `tkpack`. One does not use `tkpack` and `tkgrid` to manage children of the same parent, but these are children of `tbFrame`, not `f`.

```
tkpack(makeIcon(tbFrame, "ok"), side="left")
tkpack(makeIcon(tbFrame, "quit"), side="left")
tkpack(makeIcon(tbFrame, "cancel"), side="left")
```

These two bindings change the state of the buttons as the mouse hovers over it:

```
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 ...")
}
```

```
function (W)
{
    if (as.character(tkcget(W, "-style")) == "Toolbar.TButton")
        cat("... do something for toolbar buttons ...")
}
```

Example 17.5: Using `tkgrid` to layout a calendar
This example shows how to create a simple calendar using a grid layout. (No such widget is standard with `tcltk`.) The following relies on some date functions in the `ProgGUIInR` package.

```
makeMonth <- function(w, year, month) {
  ## add headers
  days <- c("S","M","T","W","Th","F","S")
  sapply(1:7, function(i) {
    l <- ttklabel(w, text=days[i])
    tkgrid(l, row=0, column=i-1, sticky="")
  })
  ## add days
  sapply(seq_len(days.in.month(year, month)),  function(day) {
    l <- ttklabel(w, text=day)
    tkgrid(l, row=1 + week.of.month(year, month, day),
```

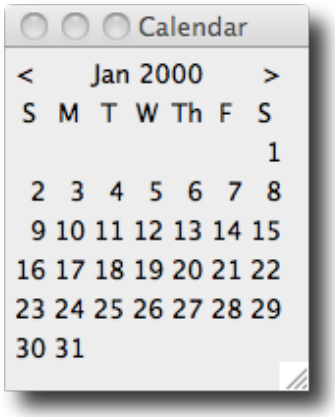

Figure 17.5: A monthly calendar illustrating various layouts.

```
            column=day.of.week(year, month, day),
            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.

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both", side="top")
cframe <- ttkframe(f)
calframe <- ttkframe(f)
tkpack(cframe, fill="x", side="top")
tkpack(calframe, 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.

```
prevb <- ttklabel(cframe, text="<")
nextb <- ttklabel(cframe, text=">")
curmo <- ttklabel(cframe)
#
tkpack(prevb, side="left", anchor="w")
tkpack(curmo, side="left", anchor="center", expand=TRUE)
tkpack(nextb, side="left", anchor="e")
```

The `setMonth` 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 `makeMonth` creates the calendar. Packing in the frame after adding its child components makes the GUI seem much more responsive.

```
setMonth <- function() {
  tkpack("forget", calframe)
  calframe <<- ttkframe(f)
  makeMonth(calframe, year, month)
  tkconfigure(curmo,                        # month label
              text=sprintf("%s %s", month.abb[month], year))
  tkpack(calframe)
}
setMonth()                                  # 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(prevb, "<Button-1>", function() {
  if(month > 1) {
    month <<- month - 1
  } else {
    month <<- 12; year <<- year - 1
  }
  setMonth()
})
```

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(tkcget(W, "-text"))
  if(!is.na(day))
    print(sprintf("You selected: %s/%s/%s", month, day, year))
})
```

17.4 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 one does not use the geometry managers to manage them. Instead, the add command is used to add a child component. For example:

```
w <- tktoplevel(); tkwm.title(w, "Paned window example")
pw <- ttkpanedwindow(w, orient="horizontal")
tkpack(pw, expand=TRUE, fill="both")
left <- ttklabel(pw, text="left")
right <- ttklabel(pw, text="right")
#
tkadd(pw, left, weight=1)
tkadd(pw, right, weight=2)
```

When resizing which child gets the space is determined by the associated weight, specified as an integer. The default uses even weights. Unlike GTK+ more than two children are allowed.

Forget The subcommand *ttkpanedwindow* forget can be used to unmanage a child component. For the paned window, we have no convenience function, so we call as follows:

```
tcl(pw, "forget", right)
tkadd(pw, right, weight=2) ## 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 specifying, as there can be more than one. Counting starts at 0. The value for sashpos is in terms of pixel width (or height) of the paned window. The width can be returned and used as follows:

```
width <- as.integer(tkwinfo("width", pw))  # or "height"
tcl(pw, "sashpos", 0, floor(0.75*width))
```

```
<Tcl> 45
```

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

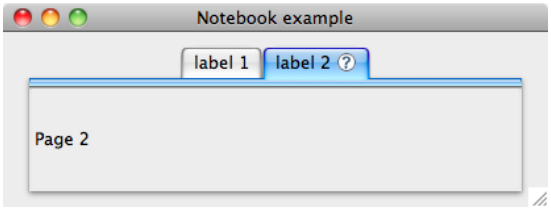

Figure 17.6: A basic notebook under Mac OS X

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, either as an R object or an ID; or in terms of *x*-*y* coordinates in the form `"@x,y"` (likely found through a binding).

To illustrate, if `nb` is a `ttknotebook` object, then these commands would add pages (cf. Figure 17.6):

```
iconFile <- system.file("images",paste("help","gif",sep="."),
                        package="gWidgets")
iconName <- "::tcl::helpIcon"
tkimage.create("photo", iconName, file = iconFile)
#
l2 <- ttklabel(nb, text="Page 2")
tkadd(nb, l2, sticky="nswe", text="label 2",
    image=iconName, compound="right")
## put l1 first (a tabID of 0); use tkinsert
l1 <- ttklabel(nb, text="Page 1")
tkinsert(nb, 0, l1, sticky="nswe", text="label 1")
```

There are several useful subcommands to extract information from the notebook object. For instance, `index` to return the page index (0-based), `tabs` to return the page IDs, `select` to select the displayed page, and `forget` to remove a page from the notebook. (There is no means to place close icons on the tabs.) Except for `tabs`, these require a specification of a tab ID.

```
tcl(nb, "index", "current")           # current page for tabID
```

```
<Tcl> 1
```

```
length(as.character(tcl(nb,"tabs")))  # number of pages
```

```
[1] 2
```

```
tcl(nb, "select", 0)          # select viewable page by index
tcl(nb, "forget", l1)         # "forget" removes a page
tcl(nb, "add", l1)            # 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", nb)
```

If enabled, the shortcuts such as control-tab to move to the next tab are implemented. If new pages are added or inserted with the option `underline`, which takes an integer value (0-based) specifying which character in the label is underlined, then a keyboard accelerator is added for that letter.

Bindings Beyond the usual events, the notebook widget also generates a `<<NotebookTabChanged>>` virtual event after a new tab is selected.

The notebook container in Tk has a few limitations. For instance, there is no graceful management of too many tabs, as there is with GTK+, as well there is no easy way to implement close buttons as an icon, as in Qt.

Draft version, do not circulate – February 5, 2011

18

Tcl/Tk: 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 these standard dialogs, then cover the basic controls before finishing up with the more involved `tktext`, `ttktreeview`, and `tkcanvas` widgets.

18.1 Dialogs

Modal dialogs

The `tkmessageBox` constructor can be used to create simple modal dialogs allowing a user to confirm an action, using 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 messageBox dialog has an `icon` argument. The default icon is `"info"` but could also be one of `"error"`, `"question"`, or `"warning"`. The buttons used are specified 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:

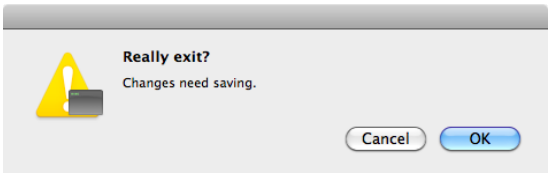

Figure 18.1: A basic modal dialog constructed by `tkmessageBox`.

```
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 toplevel 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:

```
message <- "We care ..."
dlg <- tktoplevel(); tkwm.withdraw(dlg)
tkwm.overrideredirect(dlg, TRUE)   # no decoration
f <- ttkframe(dlg, padding=5)
tkpack(f, expand=TRUE, fill="both")
tkpack(ttklabel(f, text=message), pady=5)
```

We will use `tkwait.variable` which waits for a change to variable, in this case `flag` defined next. In the button's command we release the window then change this value, ending the wait.

```
flag <- tclVar("")
tkpack(ttkbutton(f, text="dismiss", command=function() {
  tkgrab.release(dlg)
  tclvalue(flag) <- "Destroy"
}))
```

Now we show the window and wait on the `flag` variable to change.

```
tkwm.deiconify(dlg)
tkwait.variable(flag)
```

When the flag is changed, the flow returns to the program. Here we print a message then destroy the dialog.

```
print("Thanks")
```

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

```
tkdestroy(dlg)
```

File and directory selection

Tk provides constructors for selecting a file, for selecting a directory or for specifying a filename when saving. These are implemented by `tkgetOpenFile`, `tkchooseDirectory`, and `tkgetSaveFile` respectively. Each of these can be called with no argument, and each returns a `tclObj` object. The value is empty when there is no selection made.

The dialog will appear in a relationship with a toplevel window if the argument `parent` is specified The `initialdir` and `initialfile` can be used to specify the initial values in the dialog. The `defaultextension` argument can be used to specify a default extension for the file.

When browsing for files, it can be convenient to filter the available file types that can be selected. The `filetypes` argument is used for this task. However, the file types are specified using Tcl brace-notation, not R code. For example, to filter out various image types, one could have

```
tkgetOpenFile(filetypes = paste(
                "{{jpeg files} {.jpg .jpeg} }",
                "{{png files} {.png}}",
                "{{All files} {*}}", sep=" ")) # needs space
```

Extending this is hopefully clear from the pattern above.

Example 18.1: A "File" menu
To illustrate, a simple example for a file menu could be:

```
w <- tktoplevel(); tkwm.title(w, "File menu example")
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
tkadd(fileMenu,"command",label="Source file...",
      command= function() {
        fName <- tkgetOpenFile(filetypes=
                        "{{R files} {.R}} {{All files} *}")
        if(file.exists(fName <- as.character(fName)))
           source(tclvalue(fName))
      })
tkadd(fileMenu, "command", label="Save workspace as...",
      command=function() {
        fName <- tkgetSaveFile(defaultextension="Rsave")
        if(nchar(fname <- as.character(fName)))
          save.image(file=fName)
      })
tkadd(fileMenu, "command", label="Set working directory...",
      command=function() {
        dName <- tkchooseDirectory()
        if(nchar(dName <- as.character(dName)))
          setwd(dName)
      })
```

Choosing a color

Tk provides the command `tk_chooseColor` to construct a dialog for selection of a color by RGB value. There are three optional arguments `initialcolor` to specify an inital 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 quantitles. There is no constructor in `tcltk`, but one can use the dialog as follows:

```
w <- tktoplevel(); tkwm.title(w, "Select a color")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
colorWell <- tkcanvas(f, width=40, height=16,
                      background="#ee11aa",
                      highlightbackground="#abababab")
tkpack(colorWell)
tkpack(ttklabel(f, text="Click color to change"))
#
tkbind(colorWell,"<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)
})
```

18.2 Selection Widgets

This section covers the many different ways to present data for the user to select a value. The widgets can use Tcl variables to refer to the value that is displayed or that the user selects. Recall, these were constructed through `tclVar` and manipulated through `tclvalue`. For example, a logical value can be stored as

```
value <- tclVar(TRUE)
tclvalue(value) <- FALSE
tclvalue(value)
```

```
[1] "0"
```

As `tclvalue` coerces the logical into the character string `"0"` or `"1"`, some coercion may be desired.

Checkbutton

The `ttkcheckbutton` constructor returns a check button object. The checkbuttons value (`TRUE` or `FALSE`) is linked to a Tcl variable which can be specified using a logical value. The checkbutton label can also be specified through a Tcl variable using the `textvariable` option. Alternately, as with the `ttklabel` constructor, the label can be specified through the `text`

option. As well, one 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 `f` is a frame, we can create a new check button with the following:

```
value <- tclVar(TRUE)
callback <- function() print(tclvalue(value))   # uses global
labelVar <- tclVar("check button label")
cb <- ttkcheckbutton(f, variable=value,
                     textvariable=labelVar, command=callback)
tkpack(cb)
```

To avoid using a global variable is not trivial here. There is no easy way to pass user data through to the callback, and there is no easy way to get the R object from the values passed through the % substitution values. The variable holding the value can be found through

```
tkcget(cb, "variable"=NULL)
```

```
<Tcl> ::RTcl7
```

But then, one needs a means to lookup the variable from this id. Here is a wrapper for the `tclVar` function and a lookup function that use an environment created by the `tcltk` package in place of a global variable.

```
ourTclVar <- function(...) {
  var <- tclVar(...)
  .TkRoot$env[[as.character(var)]] <- var
  var
}
## lookup function
getTclVarById <- function(id) {
  .TkRoot$env[[as.character(id)]]
}
```

Assuming we used `ourTclVar` above, then the callback above could be defined to avoid a global variable by:

```
callback <- function(W) {
  id <- tkcget(W, "variable"=NULL)
  print(getTclVarById(id))
}
```

In Section 18.2 we demonstrate how to encapsulate the widget and its variable in a reference class so that one need not worry about scoping rules to reference the variable.

A toggle button By default the widget draws with a check box. Optionally the widget can be drawn as a button, which when depressed indicates a `TRUE` state. This is done by using the style `Toolbutton`, as in:

```
tkconfigure(cb, style="Toolbutton")
```

The "Toolbutton" style is for placing widgets into toolbars.

Radio Buttons

Radiobuttons are basically differently styled checkbuttons linked through a shared Tcl variable. Each radio button is constructed through the `ttkradiobutton` 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 labels, the radio button labels may be specified through a text variable or the `text` option, in which case, as with a `ttklabel`, an image may also be incorporated through the `image` and `compound` options. In Tk the placement of the buttons is managed by the programmer.

This small example shows how radio buttons could be used for selection of an alternative hypothesis, assuming `f` is a parent container.

```
values <- c("less", "greater", "two.sided")
var <- tclVar(values[3])                # initial value
callback <- function() print(tclvalue(var))
sapply(values, function(i) {
  rb <- ttkradiobutton(f, variable=var,
                       text=i, value=i,
                       command=callback)
  tkpack(rb, side="top", anchor="w")
})
```

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. Like radiobuttons and checkbuttons, 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 one to coerce the results to the desired class.)

Unlike GTK+ and Qt there is no option to include images in the displayed text. One can adjust the alignment through the `justify` options. By default, a user can add in additional values through the entry widget part of the combo box. The `state` option controls this, with the default `"normal"` and the value `"readonly"` as an alternative.

To illustrate, again suppose `f` is a parent container. Then we begin by defining some values to choose from and a Tcl variable.

```
values <- state.name
var <- tclVar(values[1])                # initial value
```

The constructor call is as follows:

```
cb <- ttkcombobox(f,
                  values=values,
                  textvariable=var,
                  state="normal",       # or "readonly"
                  justify="left")
tkpack(cb)
```

The possible values the user can select from can be configured after construction through the `values` option:

```
tkconfigure(cb, values=tolower(values))
```

There is one case where the above won't work: when there is a single value and this value contains spaces. In this case, one can coerce the value to be of class `tclObj`:

```
tkconfigure(cb, 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 value using the *ttkcombobox* `set` sub command through `tkset` or by index (0-based) using the *ttkcombobox* `current` sub command.

```
tclvalue(var) <- values[2]              # using tcl variable
tkset(cb, values[4])                    # by value
tcl(cb, "current", 4)                   # by index
```

Getting the value One can retrieve the selected object in various ways: from the Tcl variable. Additionally, the *ttkcombobox* `get` subcommand can be used through `tkget`.

```
tclvalue(var)                             # TCL variable
```

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

```
tkget(cb)                                 # get subcommand
```

```
<Tcl> california
```

```
tcl(cb, "current")                        # 0-based index
```

```
<Tcl> 4
```

Events The virtual event <<ComboboxSelected>> occurs with selection. When the combo box may be edited, a user may expect some action when the return key is pressed. This triggers a <Return> event. To bind to this event, one can do something like the following:

```
tkbind(cb, "<Return>", function(W) {
  val <- tkget(W)
  cat(as.character(val), "\n")
})
```

For editable combo boxes, the widget also supports some of the `ttkentry` commands discussed in Section 18.3.

Scale widgets

The `ttkscale` constructor to produce a themeable scale (slider) control is missing [1]. You can define your 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`, for a non-themeable slider, has the option `resolution` to set that. Additionally, the themeable slider does not have any label or tooltip indicating its current value.

As a workaround, 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. We write this using an R reference class.

```
Slider <-
  setRefClass("TtkSlider",
     fields=c("frame", "widget", "v", "x"),
     methods=list(
       initialize=function(parent, x) {
         x <<- x
         v <<- tclVar(1)
         frame <<- ttkframe(parent)
         widget <<- ttkscale(frame, from=1, to=length(x),
                             variable=v, orient="horizontal")
         #
         tkgrid(widget, row=0, column=0, columnspan=3,
                sticky="we")
         tkgrid(ttklabel(frame, text=x[1]),
                row=1, column=0)
```

[1] As of the version of `tcltk` accompanying R 2.12.0

```
          tkgrid(ttklabel(frame, text=x[length(x)]),
                 row=1, column=2)
          tkgrid.columnconfigure(frame, 1, weight=1)
          #
          .self
        },
        get_value=function() x[as.numeric(tclvalue(v))],
        set_value=function(value) {
          "Set value. Value must be in x"
          ind <- match(value, x)
          if(!is.na(ind)) {
            v_local <- v
            tclvalue(v_local) <- ind
          }
        }
        ))
```

To use this, we have:

```
w <- tktoplevel()
x <- seq(0,1,by=0.05)
s <- Slider$new(parent=w, x=x)
tkpack(s$frame, expand=TRUE, fill="x")
#
s$set_value(0.5)
print(s$get_value())
```

```
[1] 0.5
```

As seen, 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` function (using the *ttkscale* `get` and *ttkscale* `set` sub commands) can be used to get and set the value shown. They are used in the same manner as the same-named subcommands for a combo box.

Again, the `command` option can be used to specify a callback for when the slider is manipulated by the user. E.g.:

```
tkconfigure(s$widget, command=function(...) {
  print(s$get_value())
})
```

For this widget, the callback is passed a value which we ignore above.

Spin boxes

In Tk version 8.5 there is no themeable spinbox widget. In Tk the `spinbox` command produces a non-themeable spinbox. Again, there is no direct `tkspinbox` constructor, but one can be defined with:

```
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 spinbox 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 either direction through the 50 states of the U.S.

```
w <- tktoplevel()
```

```
sp <- tkspinbox(w, values=state.name, wrap=TRUE)
```

Whereas, this invocation will allow scrolling through a numeric sequence:

```
sp1 <- tkspinbox(w, from=1, to=10, increment=1)
```

```
tkpack(sp)
tkpack(sp1)
```

The basic options to set the range for a numeric spinbox are `from`, `to`, and `increment`. The `textvariable` option can be used to link the spinbox to a Tcl variable. As usual, this allows the user to easily get and set the value displayed. Otherwise, the `tkget` and `tkset` functions may be used for these tasks.

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 spinbox displays character data and is in the `"normal"` state, the widget can be controlled like the entry widget of Section 18.3.

Example 18.2: 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 18.2.

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)
```

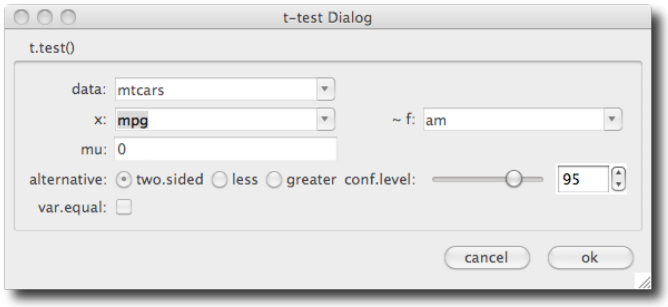

Figure 18.2: A dialog to collect values for a t test.

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.

```
lf <- ttklabelframe(f, text="t.test()", padding=10)
tkpack(lf, expand=TRUE, fill="both", padx=5, pady=5)
```

This helper function simplifies the task of adding a label.

```
putLabel <- function(parent, text, row, column) {
  label <- ttklabel(parent, text=text)
  tkgrid(label, row=row, column=column, sticky="e")
}
```

Our first widget will be one to select a data frame. For this, a combo box is used, although if a large number of data frames are a possibility, a different widget may be better suited. The `getDfs` function is not shown, but simply returns the names of all data frames in the global environment. Also not shown are two similar calls to create combo boxes `xCombo` and `fCombo` which allow the user to specify parts of a formula.

```
putLabel(lf, "data:",0,0)
dataCombo <- ttkcombobox(lf, state="readonly", values=getDfs(),
                         textvariable=e$data)
tkgrid(dataCombo, row=0, column=1, sticky="ew", padx=2)
tkfocus(dataCombo)                          # give focus
```

We jump ahead and use a `ttkentry` widget for the user to specify a mean. For this purpose, the use is straightforward.

```
putLabel(lf, "mu:", 2, 0)
muCombo <-  ttkentry(lf,  textvariable=e$mu)
tkgrid(muCombo, row=2, column=1, sticky="ew", padx=2)
```

The selection of an alternative hypothesis is a natural choice for a combo box or a radio button group, we use the latter.

```
putLabel(lf, "alternative:", 3, 0)
rbFrame <- ttkframe(lf)
sapply(c("two.sided","less","greater"), function(i) {
  rb <- ttkradiobutton(rbFrame, variable=e$alternative,
                       text=i, value=i)
  tkpack(rb, side="left")
})
tkgrid(rbFrame, row=3, column=1, sticky="ew", padx=2)
```

Here we use two widgets to specify the confidence level. The slider is quicker to use, but less precise than the spinbox. By sharing a text variable, the widgets are automatically synchronized.

```
putLabel(lf, "conf.level:", 3, 2)
confFrame <- ttkframe(lf)
tkgrid(confFrame, row=3, column=3, columnspan=2,
       sticky="ew", padx=2)
confScale <- ttkscale(confFrame, from=75, to=100,
                      variable=e$conf.level)
tkpack(confScale, expand=TRUE, fill="y", side="left")
confSpin <- tkspinbox(confFrame, from=75, to=100, increment=1,
                      textvariable=e$conf.level, width=5)
tkpack(confSpin, side="left")
```

A checkbox is used to set the binary variable for `var.equal`

```
putLabel(lf, "var.equal:", 4, 0)
veCheck <- ttkcheckbutton(lf, variable=e$var.equal)
tkgrid(veCheck, row=4, column=1, stick="w", padx=2)
```

When assigning grid weights, we don't want the labels (columns 0 and 2) to expand the same way we want the other columns to do, so we assign different weights.

```
tkgrid.columnconfigure(lf, 0, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
tkgrid.columnconfigure(lf, 2, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
```

The dialog has standard cancel and ok buttons.

```
bf <- ttkframe(f)
cancel <- ttkbutton(bf, text="cancel")
ok <- ttkbutton(bf, text="ok")
#
tkpack(bf, fill="x", padx=5, pady=5)
tkpack(ttklabel(bf, text=" "), expand=TRUE, fill="y",
       side="left")                        # add a spring
```

```
tkpack(cancel, padx=6, side="left")
tkpack(ok, padx=6, side="left")
```

For the `ok` 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 button's commands when they have the focus and return is pressed.

```
tkconfigure(ok, command=runTTest)
tkconfigure(cancel, command=function() tkdestroy(w))
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 effect its display. A such, we define a function, `updateUI`, which does two basic things: it searches for new data frames and it adjusts the controls depending on the current state.

```
updateUI <- function() {
  dfName <- tclvalue(e$data)
  curDfs <- getDfs()
  tkconfigure(dataCombo, values=curDfs)
  if(!dfName %in% curDfs) {
    dfName <- ""
    tclvalue(e$data) <- ""
  }

  if(dfName == "") {
    ## 3 ways to disable needed !
    x <- list(xCombo, fCombo, muCombo,  confScale, veCheck, ok)
    sapply(x, function(W) tcl(W, "state", "disabled"))
    sapply(as.character(tkwinfo("children", rbFrame)),
           function(W) tcl(W, "state", "disabled"))
    tkconfigure(confSpin, state="disabled")
  } else {
    ## enable univariate, ok
    sapply(list(xCombo,  muCombo, confScale, ok),
           function(W) tcl(W, "state", "!disabled"))
    sapply(as.character(tkwinfo("children", rbFrame)),
           function(W) tcl(W, "state", "!disabled"))
    tkconfigure(confSpin, state="normal")

    df <- get(dfName, envir=.GlobalEnv)
    numVars <- getNumericVars(df)
    tkconfigure(xCombo, values=numVars)
    if(! tclvalue(e$x) %in% numVars)
      tclvalue(e$x) <- ""

    ## bivariate
```

```
    availFactors <- getTwoLevelFactor(df)
    sapply(list(fCombo, veCheck),
           function(W) {
             val <- if(length(availFactors)) "!" else ""
             tcl(W, "state", sprintf("%sdisabled", val))
           })
    tkconfigure(fCombo, values=availFactors)
    if(!tclvalue(e$f) %in% availFactors)
      tclvalue(e$f) <- ""

        }
}
updateUI()
tkbind(dataCombo, "<<ComboboxSelected>>", updateUI)
```

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 only call if the available data frames have been changed. Here is one way to arrange that:

```
require(digest)
create_function <- function() {
  .dfs <- digest(getDfs())
  f <- function(...) {
    if((val <- digest(getDfs())) != .dfs) {
      .dfs <<- val
      updateUI()
    }
    return(TRUE)
  }
}
```

Then to create a task callback we have

```
id <- addTaskCallback(create_function())
```

18.3 Text widgets

Tk provides both single- and multi-line text entry widgets. The section describes both and introduces scrollbars which are often desired for multi-line text entry.

Entry Widgets

The `ttkentry` constructor provides a single line text entry widget. The widget can be associated with a Tcl variable at construction to facilitate

getting and setting the displayed values through its argument `textvariable`. The width of the widget can be adjusted at construction time through the `width` argument. This takes a value for the number of characters to be displayed, assuming average-width characters. The text alignment can be set through the `justify` argument taking values of `"left"` (the default), `"right"` and `"center"`. For gathering passwords, the argument `show` can be used, such as with `show="*"`, to show asterisks in place of all the characters.

The following constructs a basic example

```
eVar <- tclVar("initial value")
e <- ttkentry(w, textvariable=eVar)
tkpack(e)
```

We can get and set values using the Tcl variable.

```
tclvalue(eVar)
```

```
[1] "initial value"
```

```
tclvalue(eVar) <- "set value"
```

The `get` command can also be used.

```
tkget(e)
```

```
<Tcl> set value
```

Indices The entry widget uses an index to record the different positions within the entry box. This index can be a number (0-based), an x-coordinate of the value (`@x`), or one of the values `"end"` and `"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(e, "end", "new text")
```

Or, we can delete a range of text, in this case the first 4 characters, using *ttkentry* `delete`. The first value is the left most index to delete (0-based), the second value the index to the right of the last value deleted.

```
tkdelete(e, 0, 4)
```

The *ttkentry* `icursor` command can be used to set the cursor position to the specified index.

```
tkicursor(e, 0)                          # move to beginning
```

Finally, we note that the selection can be adjusted using the *ttkentry* `selection range` subcommand. This takes two indices. Like `delete`, the first index specifies the first character of the selection, the second indicates the character to the right of the selection boundary. The following example would select all the text.

```
tkselection.range(e, 0, "end")
```

The *ttkentry* `selection clear` subcommand clears the selection and *ttkentry* `selection present` signals if a selection is currently made.

Events Several useful events include <KeyPress> and <KeyRelease> for key presses and <FocusIn> and <FocusOut> for focus events.

Example 18.3: Putting in default text

In this example we show how to augment the `ttkentry` 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 their mouse on it, the initial message clears and the user can type in their value.

We use an R reference class for our programming, as it nicely allows us to encapsulate the entry widget, its tclvariable and the initial message. For formatting purposes, we define the methods first, then the class.

We begin with two methods to get and set the text. The field `v` will hold our tclvariable.

```
get_text <- function() {
  "Get the text value"
  if(!is_init_msg())
    as.character(tclvalue(v))
  else
    ""
}
#
set_text <- function(text, hide=TRUE) {
  "Set text into widget"
  if(hide)
    hide_init_msg()
  v_local <- v
  tclvalue(v_local) <- text
}
```

For the initial message we define three methods. The first checks if the current state is the initial message. Rather than use a flag, we just check if the text matches the initial message, which is stored in the field `init_msg`.

```
is_init_msg <- function() {
  "Is the init text showing?"
```

```
  as.character(tclvalue(v)) == init_msg
}
```

To indicate to the user that the initial message is not the current text, we define a style for when the initial message is being shown. It simply sets the foreground (text) color to gray.

```
.Tcl("ttk::style configure Gray.TEntry -foreground gray")
```

Now our two methods to hide and show the initial message are defined. Outside of changing the text, we adjust the style option accordingly.

```
hide_init_msg <- function() {
  "Hide the initial text"
  if(is_init_msg()) {
    tkconfigure(e, style="TEntry")
    set_text("", hide=FALSE)
  }
}
#
show_init_msg <- function() {
  "Show the initial text"
  tkconfigure(e, style="Gray.TEntry")
  set_text(init_msg, hide=FALSE)
}
```

To switch between the initial message and the entry area we make bindings to the focus in and focus out events for the entry widget. The focus out will show the initial message if there is no text specified.

```
add_bindings <- function() {
  "Add focus bindings to make this work"
  tkbind(e, "<FocusIn>", hide_init_msg)
  tkbind(e, "<FocusOut>", function() {
    if(nchar(get_text()) == 0)
      show_init_msg()
  })
}
```

Now to create the reference class to hold our widget. First we set as old classes the class for the entry widget and the tclvariable. This allows us to use these as fields in the class.

```
setOldClass("tkwin"); setOldClass("tclVar")
```

Our entry class has a straightforward initialization method, otherwise we simply piece together the components we have just defined.

```
ttkEntry <-
  setRefClass("TtkEntry",
        fields=list(
```

```
          e="tkwin", v="tclVar",
          init_msg="character"
          ),
        methods=list(
          initialize=function(parent, text, init_msg="") {
            v <<- tclVar()
            e <<- ttkentry(parent, textvariable=v)
            init_msg <<- init_msg
            if(missing(text))
              show_init_msg()
            else
              set_text(text)
            add_bindings()
            .self
          },
          get_text=get_text,
          set_text=set_text,
          is_init_msg=is_init_msg,
          hide_init_msg=hide_init_msg,
          show_init_msg=show_init_msg,
          add_bindings=add_bindings
          )
        )
```

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 that below.

```
w <- tktoplevel()
e <- ttkEntry$new(parent=w, init_msg="type value here")
tkpack(e$e)
#
b <- ttkbutton(w, text="focus out onto this",
               command=function() {
                 print(e$get_text())
               })
tkpack(b)
```

Example 18.4: Using validation for dates

As previously mentioned, there is no native calendar widget in `tcltk`. This example shows how one can use the validation framework for entry widgets to check that user-entered dates conform to an expected format.

Validation happens in a few steps. A validation command is assigned to some event. This call can come in two forms. Prevalidation is when a change is validated prior to being committed, for example when each key is pressed. Revalidation is when the value is checked after it is sent to

be committed, say when the entry widget loses focus or the enter key is pressed.

When a validation command is called it should check whether the current state of the entry widget is valid or not. If valid, it returns a value of `TRUE` and `FALSE` otherwise. These need to be Tcl Boolean values, so in the following, the command `tcl("eval","TRUE")` (or `tcl("eval", "FALSE")`) is used. If the validation command returns `FALSE`, then a subsequent call to the specified invalidation command is made.

For each callback, a number of substitution values are possible, in addition to the standard ones such as `W` to refer to the widget. These are: `d` for the type of validation being done: 1 for insert prevalidation, 0 for delete prevalidation, or -1 for revalidation; `i` for the index of the string to be inserted or deleted or -1; `P` for the new value if the edit is accepted (in prevalidation) or the current value in revalidation; `s` for the value prior to editing; `S` for the string being inserted or deleted, `v` for the current value of `validate` and `V` for the condition that triggered the callback.

In the following callback definition we use `W` so that we can change the entry text color to black and format the data in a standard manner and `P` to get the entry widget's value just prior to validation.

To begin, we define some patterns for acceptable date formats.

```
datePatterns <- c()
for(i in list(c("%m","%d","%Y"),        # U.S. style
              c("%m","%d","%y"))) {
  for(j in c("/","-"," ") )
    datePatterns[length(datePatterns)+1] <-
      paste(i,sep="", collapse=j)
}
```

Our callbacks set the color to black or red, depending on whether we have a valid date. First our validation command.

```
isValidDate <- function(W, P) { # P is the current value
  for(i in datePatterns) {
    date <- try( as.Date(P, format=i), silent=TRUE)
    if(!inherits(date, "try-error") && !is.na(date)) {
      tkconfigure(W, foreground="black")  # 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.

```
indicateInvalidDate <- function(W) {
```

```
  tkconfigure(W,foreground="red")
  tcl("expr","TRUE")
}
```

The `validate` argument is used to specify when the validation command should be called. This can be a value of `"none"` for validation when called through the `validation` command; `"key"` for each key press; `"focusin"` for when the widget receives the focus; `"focusout"` for when it loses focus; `"focus"` for both of the previous; and `"all"` for any of the previous. We use `"focusout"` below, so also give a button widget so that the focus can be set elsewhere.

```
e <- ttkentry(f, validate="focusout",   # f some parent
              validatecommand=isValidDate,
              invalidcommand=indicateInvalidDate)
tkpack(e, side="left")
b <- ttkbutton(f,text="click")       # something to focus on
tkpack(b, side="bottom")
```

Scrollbars

Tk has several scrollable widgets – those that use scrollbars. Widgets which accept a scrollbar (without too many extra steps) have the options `xscrollcommand` and `yscrollcommand`. To use scrollbars in `tcltk` requires two steps: the scrollbars must be constructed and bound to some widget, and that widget must be told it has a scrollbar. This way changes to the widget can update the scrollbar and vice versa. Suppose, `parent` is a container and `widget` has these options, then the following will set up both horizontal and vertical scrollbars. The scrollbars are defined first, as follows, using the `orient` option and a command of the following form.

```
xscr <- ttkscrollbar(parent, orient="horizontal",
                     command=function(...) tkxview(widget, ...))
yscr <- ttkscrollbar(parent, orient="vertical",
                     command=function(...) tkxview(widget, ...))
```

The `view` commands set what part of the widget is being shown.

To link the widget back to the scrollbar, the `set` command is used in a callback to the scroll command. For this example we configure the options after the widget is constructed, but this can be done at the time of construction as well. Again, the command takes a standard form:

```
tkconfigure(widget,
            xscrollcommand=function(...) tkset(xscr,...),
            yscrollcommand=function(...) tkset(yscr,...))
```

Although scrollbars can appear anywhere, the conventional place is on the right and lower side of the parent. The following adds scrollbars using

the grid manager. The combination of weights and stickiness below will have the scrollbars expand as expected if the window is resized.

```
tkgrid(widget, row=0, column=0, sticky="news")
tkgrid(yscr,row=0,column=1, sticky="ns")
tkgrid(xscr, row=1, column=0, sticky="ew")
tkgrid.columnconfigure(parent, 0, weight=1)
tkgrid.rowconfigure(parent, 0, weight=1)
```

Although a bit tedious, this gives the programmer some flexibility in arranging scrollbars. To avoid doing all this in the sequel, we turn the above into the function `addScrollbars` for subsequent usage (not shown). In base Tk, there is no means to hide scrollbars when not needed. The `tcltk2` has some code that may be employed for that.

Multi-line Text Widgets

The `tktext` widget creates a multi-line text editing widget. If constructed with no options but a parent container, the widget can have text entered into it by the user.

The text widget is not a themed widget, hence has numerous arguments to adjust its appearance. We mention a few here and leave the rest to be discovered in the manual page (along with much else). The argument `width` and `height` are there to set the initial size, with values specifying number of characters and number of lines (not pixels, to convert see Section 16.2). The actual size is font dependent, with the default for 80 by 24 characters. The `wrap` argument, with a value from `"none"`, `"char"`, or `"word"`, indicates if wrapping is to occur and if so, does it happen at any character or only a word boundary. The argument `undo` takes a logical value indicating if the undo mechanism should be used. If so, the subcommand *tktext* `edit` can be used to undo a change (or the control-z keyboard shortcut).

Indices As with the entry widget, several commands take indices to specify position within the text buffer. Only for the multi-line widget both a line and character are needed in some instances. These indices may be specified in many ways. One can use row and character numbers separated by a period in the pattern `line.char`. The line is 1-based, the column 0-based (e.g., `1.0` says start on the 1st row and first character). In general, one can specify any line number and character on that line, with the keyword `end` used to refer to the last character on the line. Text buffers may carry transient marks, in which case the use of this mark indicates the next character after the mark. Predefined marks include `end`, to specify the end of the buffer, `insert`, to track the insertion point in the text buffer were the user to begin typing, and `current`, which follows the character

closest to the mouse position. As well, pieces of text may be tagged. The format `tag.first` and `tag.last` index the range of the tag `tag`. Marks and tags are described below. If the *x-y* postion of the spot is known (through percent substitutions say) the index can be specified by postion, as `x,y`.

Indices can also be adjusted relative to the above specifications. This adjustment can be by a number of characters (`chars`), index positions (`indices`) or `lines`. For example, `insert + 1 lines` refers to 1 line under the insert point. The values `linestart`, `lineend`, `wordstart` and `wordend` are also available. For instance, `insert linestart` is the beginning of the line from the insert point, while `end -1 wordstart` and `end - 1 chars wordend` refer to the beginning and ending of the last word in the buffer. (The `end` index refers to the character just after the new line so we go back 2 steps.)

Getting text The *tktext* `get` subcommand is used to retrieve the text in the buffer. Coercion to character should be done with `tclvalue` and not `as.character` to preserve the distinction between spaces and line breaks.

```
value <- tkget(t, "1.0", "end")
as.character(value)                    # wrong way
```

```
character(0)
```

```
tclvalue(value)
```

```
[1] "\n"
```

Inserting text Inserting text can be done through the *ttktext* `insert` subcommand by specifying first the index then the text to add. One can use `\n` to add new lines.

```
tkinsert(t, "end", "more text\n new line")
```

Images and other windows can be added to a text buffer, but we do not discuss that here.

The buffer can have its contents cleared using `tkdelete`, as with `tkdelete(t, "0.0", "end")`.

Panning the buffer: tksee After text is inserted, the visible part of buffer may not be what is desired. The *ttktext* `see` sub command is used to position the buffer on the specified index, its lone argument.

tags Tags are a means to assign a name to characters within the text buffer. Tags may be used to adjust the foreground, background and font properties of the tagged characters from those specified globally at the

time of construction of the widget, or configured thereafter. Tags can be set when the text is inserted by appending to the argument list, as with

```
tkinsert(t, "end", "last words", "lastWords") # lastWords tag
```

Tags can be set after the text is added through the *tktext* `tag add` subcommand using indices to specify location. The following marks the first word with the `firstWord` tag:

```
tktag.add(t,"firstWord","1.0 wordstart", "1.0 wordend")
```

The *tktext* `tag configure` can be used to configure properties of the tagged characters, for example:

```
tktag.configure(t, "firstWord", foreground="red",
                font="helvetica 12 bold")
```

There are several other configuration options for a tag. 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 if there is a current selection, the following may be used:

```
hasSelection <- function(W) {
  ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
  length(ranges) > 1 || ranges != ""
}
```

The cut, copy and paste commands are implemented through the 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, one has

```
tcl("tk_textCut", t)
```

marks Tags mark characters within a buffer, marks denote positions within a buffer that can be modified. For example, the marks `insert` and `current` refer to the position of the cursor and the current position of the mouse. Such information can be used to provide context-sensitive popup menus, as in this code example:

```
popupContext <- function(W, x, y) {
  ## or use sprintf("@%s,$s", x, y) for "current"
  cur <- tkget(W, "current wordstart", "current wordend")
```

```
  cur <- tclvalue(cur)
  popupContextMenuFor(cur, x, y)        # some function
}
```

To assign a new mark, one uses the *tktext* `mark set` subcommand specifying a name and a position through an index. Marks refer to spaces within characters. The `gravity` of the mark can be `left` or `right`. When `right` (the default), new text inserted is to the left of the mark. For instance, to keep track of an initial insert point and the current one, the initial point (marked `leftlimit` below) can be marked with

```
tkmark.set(t,"leftlimit","insert")
tkmark.gravity(t,"leftlimit","left")    # keep onleft
tkinsert(t,"insert","new text")
tkget(t, "leftlimit", "insert")
```

```
<Tcl> new text
```

The use of the subcommand *tktext* `mark gravity` is done so that the mark attaches to the left-most character at the insert point. The rightmost one changes as more text is inserted, so would make a poor choice here.

The edit command The subcommand *tktext* `edit` can be used to undo text. As well, it can be used to test if the buffer has been modified, as follows:

```
tcl(t, "edit", "undo")                  # no output
tcl(t, "edit", "modified")              # 1 = TRUE
```

```
<Tcl> 1
```

Events The text widget has a few important events. The widget defines virtual events `<<Modified>>` and `<<Selection>>` indicating when the buffer is modified or the selection is changed. Like the single-line text widget, the events `<KeyPress>` and `<KeyRelease>` indicate key activity. The %-substitution `k` gives the keycode and `K` the key symbol as a string (`N` is the decimal number).

Example 18.5: 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 begin by defining tags for formatting purposes.

```
tktag.configure(t, "commandTag", foreground="blue",
                font="courier 12 italic")
tktag.configure(t, "outputTag", font="courier 12")
tktag.configure(t, "errorTag", foreground="red",
                font="courier 12 bold")
```

The following function does the work of evaluating a command chunk then inserting the values into the text buffer, using the different markup tags specified above to indicate commands from output.

```
evalCmdChunk <- function(t, cmds) {

  cmdChunks <- try(parse(text=cmds), silent=TRUE)
  if(inherits(cmdChunks,"try-error")) {
    tkinsert(t, "end", "Error", "errorTag") # add markup tag
  }

  for(cmd in cmdChunks) {
    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(t, "end", command, "commandTag")
    tkinsert(t, "end","\n")
    ## output, should check for errors in eval!
    output <- capture.output(eval(cmd, envir=.GlobalEnv))
    output <- paste(output, collapse="\n")
    tkinsert(t, "end", output, "outputTag")
    tkinsert(t, "end","\n")
  }
}
```

We envision this as a piece of a larger GUI which generates the commands to evaluate. For this example though, we make a simple GUI.

```
w <- tktoplevel(); tkwm.title(w, "Text buffer example")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
t <- tktext(f, width=80, height = 24)   # default size
addScrollbars(f, t)
```

This is how it can be used.

```
evalCmdChunk(t, "2 + 2; lm(mpg ~ wt, data=mtcars)")
```

18.4 Treeview widget

The themed treeview widget can be used to display rectangular data, like a data frame, or hierarchical data. The usage is similar for each beyond the need to indicate the hierarchical structure of a tree.

Rectangular data

The `ttktreeview` constructor creates the tree widget. There is no separate model for this widget, 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 `displaycolunms` 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 different ways. The `height` argument is used to adjust the number of visible rows. The width of the widget is determined by the combined widths of each column, whose adjustments are mentioned later.

If `f` is a frame, then the following call will create a widget with just one column showing 25 rows, like the older, non-themed, listbox widget of Tk.

```
tr <- ttktreeview(f,
                  columns=1,        # column identifier is "1"
                  show="headings",  # not "#0"
                  height=25)
addScrollbars(f, tr)                # scrollbar function
```

The treeview widget has an initial column for showing the tree-like aspect with the data. This column is referenced by `#0`. The `show` argument controls whether this column is shown. A value of `"tree"` leaves just this column shown, `"headings"` will show the other columns, but not the first, and the combined value of `"tree headings"` will display both (the default). Additionally, the treeview is a scrollable widget, so has the arguments `xscrollcommand` and `yscrollcommand` for specifying scrollbars.

Adding values Rectangular data has a row and column structure. In R, data frames are internally stored by column vectors, so each column may have its own type. The treeview widget is different, it stores all data as character data and one interacts 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 `""` for rectangular data) and an index for specifying the location of the new child amongst the previous children. The special value `"end"` indicates placement after all other children, as would a number larger than the number of children. A value of 0 or a negative value would put it at the beginning.

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 1:length(Host))
  ID <- tkinsert(tr, "", "end", values=as.tclObj(Host[i]),
                 tag=shade[i %% 2])                 # none or gray
tktag.configure(tr, "gray", background="gray95") # shade rows
```

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. 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 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 allows the use of the subcommand *ttktreeview* `tag configure` to configure the foreground color, background color, font or image of 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(tr, "heading", 1, text="Host", anchor="center")
```

The *ttktreeview* `column` subcommand can be used to adjust a column's properties including the size of the column. The option `width` is used to specify the pixel width of the column (the default is large); As the widget may be resized, one can specify the minimum column width through the option `minwidth`. When more space is allocated to the tree widget, than is requested by the columns, column with a `TRUE` value specified to the option `stretch` are resized to fill the available space. Within each column, the placement of each entry within a cell is controlled by the `anchor` option, using the compass points.

For example, this command will adjust properties of the lone column of `tr`:

```
tcl(tr, "column", 1, width=400,  stretch=TRUE, anchor="w")
```

Item IDs Each row has a unique item ID generated by the widget when a row is added. The base ID is "" (why this is specified for the value of `parent` for rectangular data). For rectangular displays, the list of all IDs may be found through the *ttktreeview* `children` sub command, which we will describe in the next section. Here we see it used to find the children of the root. As well, we show how the *ttktreeview* `index` command returns the row index.

```
children <- tcl(tr, "children", "")
(children <- head(as.character(children)))     # as.character
```

```
[1] "I001" "I002" "I003" "I004" "I005" "I006"
```

```
sapply(children, function(i) tclvalue(tkindex(tr, i)))
```

```
I001 I002 I003 I004 I005 I006
 "0"  "1"  "2"  "3"  "4"  "5"
```

Retrieving values The *ttktreeview* `item` subcommand can be used to get the values and other properties stored for each row. One specifies the item and the corresponding option:

```
x <- tcl(tr, "item", children[1], "-values") # no tkitem
as.character(x)
```

```
[1] "University of Melbourne"
```

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 instead keep the original data frame and use the index of the item to extract the value from the original.

Moving and deleting items The *ttktreeview* `move` subcommand can be used to replace a child. As with the `insert` command, a parent and an index for where the new child is to go among the existing children is 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 0, 1 or more than 1 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 6 children, we have:

```
tkselect(tr, "set", children)
```

To toggle the selection, we have:

```
tkselect(tr, "toggle", tcl(tr, "children", ""))
```

Finally, the selected IDs are returned with:

```
IDs <- as.character(tkselect(tr))
```

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.

```
callbackExample <- function(W, x, y) {
  col <- as.character(tkidentify(W, "column", x, y))
  row <- as.character(tkidentify(W, "row", x, y))
  ## now do something ...
}
```

Example 18.6: 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.

```
f0 <- ttkframe(f); tkpack(f0, fill="x")
l <- ttklabel(f0, text="filter:"); tkpack(l, side="left")
filterVar <- tclVar("")
filterEntry <- ttkentry(f0, textvariable=filterVar)
tkpack(filterEntry, side="left")
```

The treeview will only show the first three columns of the data frame, although we store the fourth which contains the URL.

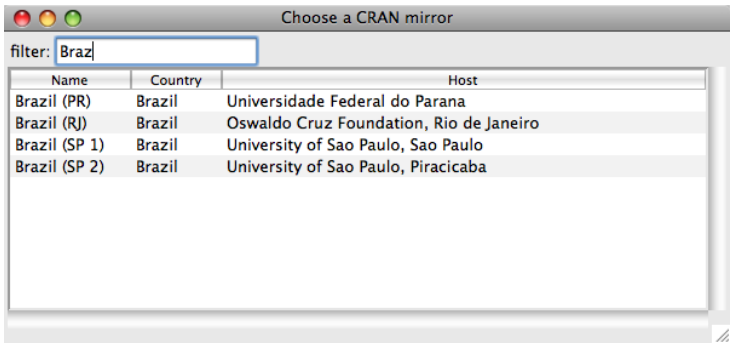

Figure 18.3: Using `ttktreeview` to show various CRAN sites. This illustration adds a search-like box to filter what repositories are displayed for selection.

```
f1 <- ttkframe(f); tkpack(f1, expand=TRUE, fill="both")
tr <- ttktreeview(f1, columns=1:ncol(df),
                  displaycolumns = 1:(ncol(df) - 1),
                  show = "headings",     # not "tree"
                  selectmode = "browse") # single selection
addScrollbars(f1, tr)
```

We configure the column widths and titles as follows:

```
widths <- c(100, 75, 400)            # hard coded
nms <- names(df)
for(i in 1:3) {
  tcl(tr, "heading", i, text=nms[i])
  tcl(tr, "column", i, width=widths[i],
      stretch=TRUE, anchor="w")
}
```

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(tr, df) {
  children <- as.character(tcl(tr, "children", ""))
  for(i in children) tcl(tr, "delete", i)    # out with old
  shade <- c("none", "gray")
  for(i in seq_len(nrow(df)))
    tcl(tr, "insert", "", "end", tag=shade[i %% 2],
        text="",
        values=unlist(df[i,]))               # in with new
  tktag.configure(tr, "gray", background="gray95")
```

[2] The model-view-controller architecture of GTK+, say, makes this task much easier, as it allows for an intermediate proxy model.

```
}
```

The initial call populates the table from the entire data frame.

```
fillTable(tr, df)
```

The filter works by grepping the user input against the host value. We bind to <KeyRelease> (and not <KeyPress>) so we capture the last keystroke.

```
curInd <- 1:nrow(df)
tkbind(filterEntry, "<KeyRelease>", function(W, K) {
  val <- tclvalue(tkget(W))
  possVals <- apply(df, 1, function(...)
                    paste(..., collapse=" "))
  ind<- grep(val, possVals)
  if(length(ind) == 0) ind <- 1:nrow(df)
  fillTable(tr, df[ind,])
})
```

This binding is for capturing a users selection through a double-click event. In the callback, we set the CRAN option then withdraw the window.

```
tkbind(tr, "<Double-Button-1>", function(W, x, y) {
  sel <- as.character(tcl(W, "identify", "row", x, y))
  vals <- tcl(W, "item", sel, "-values")
  URL <- as.character(vals)[4]          # not tclvalue
  repos <- getOption("repos")
  repos["CRAN"] <- gsub("/$", "", URL[1L])
  options(repos = repos)
  tkwm.withdraw(tkwinfo("toplevel", W))
})
```

Editing cells of a table There is no native widget for editing the cells of tabular data, as is provided by the `edit` method for data frames. The `tktable` widget (`http://tktable.sourceforge.net/`) provides such an add-on to the base Tk. We don't illustrate its usage here, as we keep to the core set of functions provided by Tk. 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

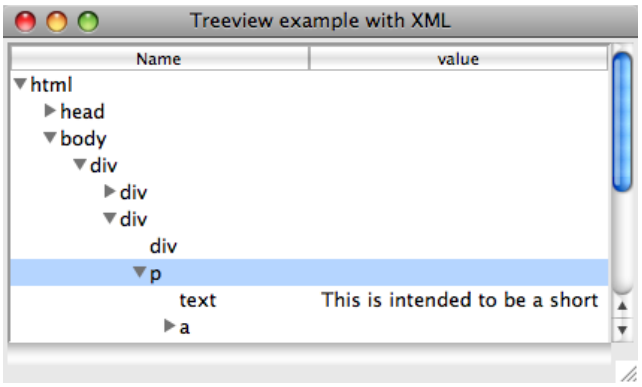

Figure 18.4: Illustration of using `ttktreeview` widget to show hierarchical data returned from parsing an HTML document with the `XML` package.

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, only instead of using the root item, `""`, as the parent item, one uses the item ID corresponding to the desired parent. If the option `open=TRUE` is specified to the `insert` subcommand, the children of the item will appear, if `FALSE`, the user can click the trigger icon to see the children. The programmer can use the *ttktreeview* `item` to configure this state. When the parent item is opened or closed, the virtual events <<TreeviewOpen>> and <<TreeviewClose>> will be signaled.

Traversal Once a tree is constructed, the programmer can traverse through the items using the subcommands *ttktreeview* `parent` *item* to get the ID for the parent of the item; *ttktreeview* `prev` *item* and *ttktreeview* `next` *item* to get the immediate siblings of the item; and *ttktreeview* `children` *item* to return the children of the item. Again, the latter one will produce a character vector of IDs for the children when coerced to character with `as.character`.

Example 18.7: 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)
```

```
fileName <- "http://www.omegahat.org/RSXML/shortIntro.html"
QT <- function(...) {}  # quiet next call
doc <- htmlTreeParse(fileName, useInternalNodes=TRUE, error=QT)
root <- xmlRoot(doc)
```

Our GUI is primitive, with just a treeview instance added.

```
tr <- ttktreeview(f, displaycolumns="#all", columns=1)
addScrollbars(f, tr)
```

We configure our column headers and set a minimum width below. Recall, the tree column is designated `"#0"`.

```
tcl(tr, "heading", "#0", text="Name")
tcl(tr, "column", "#0", minwidth=20)
tcl(tr, "heading", 1, text="value")
tcl(tr, "column", 1, minwidth=20)
```

To map the tree-like structure of the XML document into the widget, we define the following function to recursively add to the treeview instance. We only add to the `value` column (through the `values` option) when the node does not have children. We use `do.call`, as a convenience, to avoid constructing two different calls to the `insert` subcommand.

```
insertChild <- function(tr, node, parent="") {
  l <- list(tr, "insert", parent, "end", text=xmlName(node))
  children <- xmlChildren(node)
  if(length(children) == 0) {            # add in values
    values <- paste(xmlValue(node), sep=" ", collapse=" ")
    l$values <- as.tclObj(values)        # avoid split on spaces
  }
  treePath <- do.call("tcl", l)

  if(length(children))                              # recurse
    for(i in children) insertChild(tr, i, treePath)
}
insertChild(tr, root)
```

At this point, the GUI will allow one to explore the 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 one might introduce drag and drop to rearrange the rows. We begin by defining two global variables that store the row that is being dragged and a flag to indicate if a drag event is ongoing.

```
.selectedID <- ""                                   # globals
.dragging <- FALSE
```

We provide callbacks for three events: a mouse click, mouse motion and mouse release. This first callback sets the selected row on a mouse click.

```
tkbind(tr, "<Button-1>", function(W,x,y) {
  .selectedID <<- as.character(tcl(W, "identify","row", x, y))
})
```

The motion callback configures the cursor to indicate a drag event and sets the dragging flag. One might also put in code to highlight any drop areas.

```
tkbind(tr, "<B1-Motion>", function(W, x, y, X, Y) {
  tkconfigure(W, cursor="diamond_cross")
  .dragging <<-TRUE
})
```

When the mouse button is released we check that the widget we are over is indeed the tree widget. If so, we then move the rows. One can't move a parent to be a child of its own children, so we wrap the *ttktreeview* `move` sub command within `try`. The `move` command places the new value as the first child of the item it is being dropped on. If a different action is desired, the `"0"` below would need to be modified.

```
tkbind(tr, "<ButtonRelease-1>", function(W, x, y, X, Y) {
  if(.dragging && .selectedID != "") {
    w = tkwinfo("containing", X, Y)
    if(as.character(w) == as.character(W)) {
      dropID <- as.character(tcl(W, "identify","row", x, y))
      try(tkmove(W, .selectedID, dropID, "0"), silent=TRUE)
    }
  }
  .dragging <<- FALSE; .selectedID <<- "" # reset
})
```

Walking the tree Our last item of general interest is a function that shows one way to walk the structure of the treeview widget to generate a list representing the structure of the data. A potential use of this might be to allow a user to rearrange an XML document through drag and drop. The subcommand *ttktreeview* `children` proves useful here, as it is used to identify the hierarchical structure. When there are children a recursive call is made.

```
treeToList <- function(tr) {
  l <- list()
  walkTree <- function(child, l) {
    l$name <- tclvalue(tcl(tr,"item", child, "-text"))
    l$value <- as.character(tcl(tr,"item", child, "-values"))
    children <- as.character(tcl(tr, "children", child))
```

```
    if(length(children)) {
      l$children <- list()
      for(i in children)
        l$children[[i]] <- walkTree(i, list()) # recurse
    }
    return(l)
  }
  walkTree("", l)
}
```

18.5 Menus

Menu bars and popup menus in Tk are constructed with `tkmenu`. The `parent` argument depends on what the menu is to do. A toplevel menu bar, such as appears at the top of a window has a toplevel window as its parent; a submenu of a menu bar uses the parent menu; and a popup menu uses a widget. The menu widget in Tk has an option to be "torn off." This features was at one time common in GUIs, but now is rarely seen so it is recommended that this option be disabled. The `tearoff` option can be used at construction time to override the default behavior. Otherwise, the following command will do so globally:

```
tcl("option","add","*tearOff", 0)    # disable tearoff menus
```

A toplevel menu bar is attached to a top-level window using `tkconfigure` to set the `menu` 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 toplevel window. Testing for native Mac OS X may be done via the following function:

```
usingMac <- function()
  as.character(tcl("tk", "windowingsystem")) == "aqua"
```

The `tkpopup` function facilitates the creation of a popup menu. This function has arguments for the menu bar, and the postion where the menu should be popped up. For example, the following code will bind a popup menu, `pmb` (yet to be defined), to the right click event for a button `b`. As Mac OS X may not have a third mouse button, and when it does it refers to it differently, the callback is bound conditionally to different events.

```
doPopup <- function(X, Y) tkpopup(pmb, X, Y) # define callback
if (usingMac()) {
  tkbind(b, "<Button-2>", doPopup)      # right click
  tkbind(b, "<Control-1>", doPopup)     # Control + click
} else {
```

```
    tkbind(b, "<Button-3>", doPopup)
}
```

Adding submenus and action items Menus 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 sub-menu.

Grouping of similar items can be done through nesting, or on occasion through visual separation. The latter is implemented with the *tkmenu* `add separator` subcommand.

There are a few different types of action items that can be added:

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 one to specify a `label`, a `command` and optionally an `image` with a value for `compound` to adjust its layout. (Images are not shown in Mac OS X.) Action commands may possibly be called for different widgets, so the use of percent substitution is difficult. One can also specify that a keyboard accelerator be displayed through the option `accelerator`, but a separate callback must listen for this combination.

Check boxes 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 call back 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 the menu's labels.

Set state The `state` option is used to retrieve and set the current state of the a menu item. This value is typically `normal` or `disabled`, the latter

to indicate the item is not available. The state can be set when the item is added or configured after that fact, through the *tkmenu* `entryconfigure` command. This function needs the menu bar specified and the item specified as an index or pattern to match the labels.

Example 18.8: Simple menu example
This example shows how one might make a very simple code editor using a text-entry widget. We use the `svMisc` package, as it defines a few GUI helpers which we use.

```
library(svMisc)                          # for some helpers
showCmd <- function(cmd) writeLine(captureAll(Parse(cmd)))
```

We create a simple GUI with a top-level window containing the text entry widget.

```
w <- tktoplevel()
tkwm.title(w, "Simple code editor")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
tb <- tktext(f, undo=TRUE)
addScrollbars(f, tb)
```

We create a toplevel menu bar, `mb`, and attach it to our toplevel window, then a file and edit submenu:

```
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
#
editMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="Edit", menu=editMenu)
```

To these sub menu bars, we add action items. First a command to evaluate the contents of the buffer.

```
tkadd(fileMenu, "command", label="Evaluate buffer",
      command = function() {
        curVal <- tclvalue(tkget(tb, "1.0", "end"))
        showCmd(curVal)
      })
```

Then a command to evaluate just the current selection

```
tkadd(fileMenu, "command", label="Evaluate selection",
      state="disabled",
      command = function() {
        curSel <- tclvalue(tkget(tb, "sel.first", "sel.last"))
        showCmd(curSel)
      })
```

Finally, we end the file menu with a quit action.

```
tkadd(fileMenu, "separator")
tkadd(fileMenu, "command", label="Quit",
      command=function() tkdestroy(w))
```

The edit menu has an undo and redo item. For illustration purposes we add an icon to the undo item.

```
img <- system.file("images/up.gif", package="gWidgets")
tkimage.create("photo", "::img::undo",
                     file=img)
tkadd(editMenu, "command", label="Undo",
      image="::img::undo", compound="left",
      command = function() tcl(tb, "edit", "undo"))
tkadd(editMenu, "command", label="Redo",
      command = function() tcl(tb, "edit", "redo"))
```

We now define a function to update the user interface to reflect any changes.

```
updateUI <- function() {
  states <- c("disabled","normal")
  ## selection
  hasSelection <- function(W) {
    ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
    length(ranges) > 1 || ranges != ""
  }
  ## by index
  tkentryconfigure(fileMenu,1,
                   state=states[hasSelection(tb) + 1])
  ## undo -- if buffer modified, assume undo stack possible
  ## redo -- no good check for redo
  canUndo <- function(W) as.logical(tcl(W,"edit", "modified"))
  tkentryconfigure(editMenu,"Undo",     # by pattern
                   state=states[canUndo(tb) + 1])
  tkentryconfigure(editMenu,"Redo",
                   state=states[canUndo(tb) + 1])
}
```

We now add an accelerator entry to the menubar and a binding to the top-level window for the keyboard shortcut.

```
if(usingMac()) {
  tkentryconfigure(editMenu, "Undo", accelerator="Cmd-z")
  tkbind(w,"<Option-z>", function() tcl(tb,"edit","undo"))
} else {
  tkentryconfigure(editMenu, "Undo", accelerator="Control-u")
  tkbind(w,"<Control-u>", function() tcl(tb,"edit","undo"))
}
```

To illustrate popup menus, we define one within our text widget that will grab all functions that complete the current word, using the `Com-`

pletePlus function from the svMisc package to find the completions. The use of current wordstart and current wordend to find the word at the insertion point isn't quite right for R, as it stops at periods.

```
doPopup <- function(W, X, Y) {
  cur <- tclvalue(tkget(W, "current  wordstart",
                        "current wordend"))
  tcl(W, "tag", "add", "popup", "current  wordstart",
                               "current wordend")
  posVals <- head(CompletePlus(cur)[,1, drop=TRUE], n=20)
  if(length(posVals) > 1) {
    popup <- tkmenu(tb)                # create menu for popup
    sapply(posVals, function(i) {
      tkadd(popup, "command", label=i, command = function() {
        tcl(W,"replace", "popup.first", "popup.last", i)
      })
    })
    tkpopup(popup, X, Y)
 }}
```

For a popup, we set the appropriate binding for the underlying windowing system. For the second mouse button binding in OS X, we clear the clipboard. Otherwise the text will be pasted in, as this mouse action already has a default binding for the text widget.

```
if (!usingMac()) {
  tkbind(tb, "<Button-3>", doPopup)
} else {
  tkbind(tb, "<Button-2>", function(W,X,Y) {
    ## UNIX legacy re mouse-2 click for selection copy
    tcl("clipboard","clear",displayof=W)
    doPopup(W,X,Y)
    })      # right click
  tkbind(tb, "<Control-1>", doPopup)     # Control + click
}
```

18.6 Canvas Widget

The canvas widget provides an area to display lines, shapes, images and widgets. Methods exist to create, move and delete these objects, allowing the canvas widget to be the basis for creating interactive GUIs. The constructor tkcanvas for the widget, being a non-themeable widget, has many arguments including these standard ones: width, height, and 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 the item. The basic shape types that one can add are `"line"`, `"arc"`, `"polygon"`, `"rectangle"`, and `"oval"`. Their options specify the size using x and y coordinates. Other options allow one to specify colors, etc. The complete list is covered in the `canvas` manual page, which we refer the reader to, as the description is lengthy. In the examples, we show how to use the `"line"` type to display a graph and how to use the `"oval"` type to add a point to a canvas. Additionally, one can add text items through the `"text"` type. The first options are the x and y coordinates and the `text` option specifies the text. Other standard text options are possible (e.g., `font`, `justify`, `anchor`).

The type can also be an image object or a widget (a window object). Images are added by specifying an x and y position, possibly an anchor position, and a value for the `"image"` option and optionally, for state dependent display, specifying `"activeimage"` and `"disabledimage"` values. The `"state"` option is used to specify the current state. Window objects are added similarly in terms of their positioning, along with options for `"width"` and `"height"`. The window itself is added through the `"window"` option. An example shows how to add a frame widget.

Items and tags The `tkcanvas.create` function returns an item ID. This can be used to refer to the item at a later stage. Optionally, tags can be used to group items into common groups. The `"tags"` option can be used with `tkcreate` when the item is created, or the *tkcanvas* `addtag` subcommand can be used. The call `tkaddtag(canvas, tagName, "withtag", item)` would add the tag "`tagName`"to the `item` returned by `tkcreate`. (The `"withtag"` is one of several search specifications.) As well, if one is adding a tag through a mouse click, the call `tkaddtag(W, "tagName", "closest", x, y)` could be used with `W`, `x` and `y` coming from percent substitutions. Tags can be deleted through the *tkcanvas* `dtag` *tag* subcommand.

There are several subcommands that can be called on items as specified by a tag or item ID. For example, the *tkcanvas* `itemcget` and *tkcanvas* `itemconfigure` subcommands allow one to get and set options for a given item. The *tkcanvas* `delete` *tag_or_ID* subcommand can be used to delete an item. Items can be moved 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 one 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 18.9: Using a canvas to make a scrollable frame
This example[3] shows how to use a canvas widget to create a box container that scrolls when more items are added than will fit in the display area. The basic idea is that a frame is added to the canvas equipped with scrollbars using the *tkcanvas* `create window` subcommand.

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 child window is the appropriate width when the canvas widget resizes. The height is not adjusted, as this is controlled by the scrolling.

```
scrollableFrame <- function(parent, width= 300, height=300) {
  canvasWidget <-
    tkcanvas(parent,
             borderwidth=0, highlightthickness=0,
             width=width, height=height)
  addScrollbars(parent, canvasWidget)
  #
  gp <- ttkframe(canvasWidget, padding=c(0,0,0,0))
  gpID <- tkcreate(canvasWidget, "window", 0, 0, anchor="nw",
                   window=gp)
  tkitemconfigure(canvasWidget, gpID, width=width)
  ## update scroll region
  tkbind(gp,"<Configure>",function() {
    bbox <- tcl(canvasWidget, "bbox", "all")
    tcl(canvasWidget,"config", scrollregion=bbox)
  })
  ## adjust "window" width when canvas is resized.
  tkbind(canvasWidget, "<Configure>", function(W) {
    width <- as.numeric(tkwinfo("width", W))
    gpwidth <- as.numeric(tkwinfo("width", gp))
    if(gpwidth < width)
      tkitemconfigure(canvasWidget, gpID, width=width)
  })
  return(gp)
}
```

To use this, we create a simple GUI as follows:

```
w <- tktoplevel()
```

[3]This example is modified from an example found at `http://mail.python.org/pipermail/python-list/1999-June/005180.html`

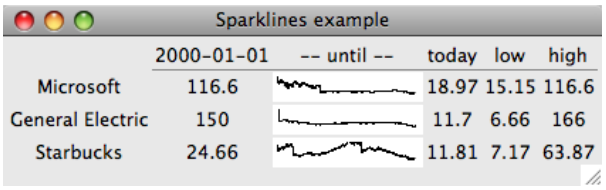

Figure 18.5: Example of embedding sparklines in a display organized using `tkgrid`. A `tkcanvas` widget is used to display the graph.

```
tkwm.title(w,"Scrollable frame example")
g <- ttkframe(w); tkpack(g, expand=TRUE, fill="both")
gp <- scrollableFrame(g, 300, 300)
```

To display a collection of available fonts requires a widget or container that could possibly show hundreds of similar values. The scrollable frame serves this purpose well (cf. Figure 16.3). 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 found from `tkfont.families` and the useful coercion to character by `as.character`.

```
fFamilies <- as.character(tkfont.families())
## skip odd named ones
fFamilies <- fFamilies[grepl("^[[:alpha:]]", fFamilies)]
for(i in seq_along(fFamilies)) {
  fName <- sprintf("::font::-%s", i)
  try(tkfont.create(fName, family=fFamilies[i], size=14),
      silent=TRUE)
  l <- ttklabel(gp, text=fFamilies[i], font=fName)
  tkpack(l, side="top", anchor="w")
}
```

Example 18.10: Using canvas objects to show sparklines
Edward Tufte, in his book *Beautiful Evidence* **?**, advocates for the use of *sparklines* – small, intense, simple datawords – to show substantial amounts of data in a small visual space. This example shows how to use a `tkcanvas` object to display a sparkline graph using a `line` object. The example also uses `tkgrid` to layout the information in a table. We could have spent more time on the formatting of the numeric values and factoring out the data download, but leave improvements as an exercise.

This function simply shortens our call to `ttklabel`. We use the global `f` (a `ttkframe`) as the parent.

```
mL <- function(label) { # save some typing
  if(is.numeric(label))
    label <- sprintf("%.2f", label)
```

```
  ttklabel(f, text=label, justify="right")
}
```

We begin by making the table header along with a toprule.

```
tkgrid(mL(""), mL("2000-01-01"), mL("-- until --"),
       mL("today"), mL("low"), mL("high"))
tkgrid(ttkseparator(f), row=1, column=1, columnspan=5,
       sticky="we")
```

This function adds a sparkline to the table. A sparkline here is just a line item, but there is some work to do, in order to scale the values to fit the allocated space. This example uses stock values, as we can conveniently employ the `get.hist.quote` function from the `tseries` package to get interesting data.

```
addSparkLine <- function(label, symbol="MSFT") {
  width <- 100; height=15               # fix width, height
  y <- get.hist.quote(instrument=symbol, start="2000-01-01",
                      quote="C", provider="yahoo",
                      retclass="zoo")$Close
  min <- min(y); max <- max(y)
  ##
  start <- y[1]; end <- tail(y,n=1)
  rng <- range(y)
  ##
  sparkLineCanvas <- tkcanvas(f, width=width, height=height)
  x <- 0:(length(y)-1) * width/length(y)
  if(diff(rng) != 0) {
    y1 <- (y - rng[1])/diff(rng) * height
    y1 <- height - y1   # adjust to canvas coordinates
  } else {
    y1 <- height/2 + 0 * y
  }
  ## make line with: pathName create line x1 y1 ... xn yn
  l <- list(sparkLineCanvas,"create","line")
  sapply(1:length(x), function(i) {
    l[[2*i + 2]] <<- x[i]
    l[[2*i + 3]] <<- y1[i]
  })
  do.call("tcl",l)

  tkgrid(mL(label),mL(start), sparkLineCanvas,
         mL(end), mL(min), mL(max), pady=2, sticky="e")
}
```

We can then add some rows to the table as follows:

```
addSparkLine("Microsoft","MSFT")
addSparkLine("General Electric", "GE")
```

```
addSparkLine("Starbucks","SBUX")
```

Example 18.11: Capturing mouse movements
This example is a stripped-down version of the `tkcanvas.R` demo that accompanies the `tcltk` package. That example shows a scatterplot with regression line. The user can move the points around and see the effect this has on the scatterplot. Here we focus on the moving of an object on a canvas widget. We assume we have such a widget in the variable `canvas`.

This following adds a single point to the canvas using an `oval` object. We add the `"point"` tag to this item, for later use. Clearly, this code could be modified to add more points.

```
x <- 200; y <- 150; r <- 6
item <- tkcreate(canvas, "oval", x - r, y - r, x + r, y + r,
                 width=1, outline="black",
                 fill="SkyBlue2")
tkaddtag(canvas, "point", "withtag", item)
```

In order to indicate to the user that a point is active, in some sense, the following changes the fill color of the point when the mouse hovers over. We add this binding using `tkitembind` so that is will apply to all point items and only the point items.

```
tkitembind(canvas, "point", "<Any-Enter>", function()
           tkitemconfigure(canvas, "current", fill="red"))
tkitembind(canvas, "point", "<Any-Leave>", function()
           tkitemconfigure(canvas, "current", fill="SkyBlue2"))
```

There are two key bindings needed for movement of an object. First, we tag the point item that gets selected when a mouse clicks on a point and update the last position of the currently selected point.

```
lastPos <- numeric(2)            # global to track position
tagSelected <- function(W, x, y) {
  tkaddtag(W, "selected", "withtag", "current")
  tkitemraise(W, "current")
  lastPos <<- as.numeric(c(x, y))
}
tkitembind(canvas, "point", "<Button-1>", tagSelected)
```

When the mouse moves, we use `tkmove` to have the currently selected point move too. As `tkmove` is parameterized by differences, we track the differences between the last position recorded and the current position.

```
moveSelected <- function(W, x, y) {
  pos <- as.numeric(c(x,y))
  tkmove(W, "selected", pos[1] - lastPos[1],
                        pos[2] - lastPos[2])
  lastPos <<- pos
```

```
}
tkbind(canvas, "<B1-Motion>", moveSelected)
```

A further binding, for the <ButtonRelease-1> event, would be added to do something after the point is released. In the original example, the old regression line is deleted, and a new one drawn. Here we simply delete the `"selected"` tag.

```
tkbind(canvas, "<ButtonRelease-1>",
       function(W) tkdtag(W,"selected"))
```

Bibliography

a. URL http://www.tcl.tk/man/tcl8.5/.

b.

Jeffrey Hobbs Brent B. Welch, Ken Jones. *Practical Programming in Tcl and Tk*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 2003.

Peter Dalgaard. The R-Tcl/Tk interface. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*. URL http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/index.html. ISSN 1609-395X.

Peter Dalgaard. A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31, September 2001. URL http://CRAN.R-project.org/doc/Rnews/.

John Fox. Extending the R Commander by "plug-in" packages. *R News*, 7(3):46–52, December 2007. URL http://CRAN.R-project.org/doc/Rnews/.

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

James Wettenhall. URL http://bioinf.wehi.edu.au/~wettenhall/RTclTkExamples/.