1

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 has been included in R since version 1.1.0. Since then, the package has been used in a number of GUI projects for R, most notably, the `Rcmdr` GUI. 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

[1] Tk has a well documented API (**?**) (`www.tcl.tk/man/tcl8.5`). There are also several books to supplement. We consulted the one by Welch, Jones and Hobbs (**?**) often in the developement of this material. The online sample chapter on Geometry Management of (**?**) was perused, as it provides a thorough discussion of that topic. In addition, the Tk Tutorial of Mark Roseman (**?**) (`www.tkdocs.com/tutorial`) provides much detail. R specific documentation include two excellent R News articles and a proceedings paper (**?**), (**?**) and (**?**) by Peter Dalgaard, the package author. A set of examples due to James 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 (**?**) by John Fox, the package author.

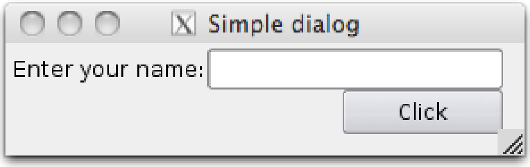

Figure 1.1: A simple dialog to collect a name for later use. This illustrates 3 basic widgets: a label, entry widget and button.

widget. We assume the Tk version is 8.5 or higher, as this was a major step forward. [2]

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. R for Windows has been bundled with the necessary Tk version for years, so there are no installation issues for that platform. For linux users, it is typically trivial to install the newer libraries and for Mac OS X users, the provided binary installations include the newer Tk libraries.

In this chapter we give an overview of Tk and R's interface to it through the `tcltk` package using the following small example of a dialog to collect a name and echo back a message (Figure 1.1). In subsequent chapters we give more detail on the various widgets provided by Tk.

```
library(tcltk)
#
w <- tktoplevel()
tkwm.title(w, "Simple dialog")
#
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
#
g <- ttkframe(f); tkpack(g)
#
l <- ttklabel(g, text="Enter your name:")
tkpack(l, side="left")
#
txtVar <- tclVar("")
txt <- ttkentry(g, textvariable=txtVar)
tkpack(txt)
#
g1 <- ttkframe(f); tkpack(g1, anchor="ne")
btn <- ttkbutton(g1, text="Click")
tkpack(btn, side="right")
```

[2] In fact, we assume version 8.5.8 which was the release accompanying R for Windows version 2.13.1.

```
#
handler = function() print(sprintf("Hello %s", tclvalue(txtVar)))
tkconfigure(btn, command=handler)
```

1.1 Interacting with Tcl

As the example above makes clear, using `tcltk` does not necessarily require knowing anything about the underlying Tk or Tcl working, though it can be useful to have a rough sense of these technologies and how `tcltk` interfaces with them. As such, we give a quick overview.

Although both are scripting languages, the basic syntax of Tcl is 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 allows arbitrary Tcl scripts to run within an R session. Tcl packages may be read in with tclRequire. [3]

The tclObj class The tcltk package creates objects with a few different classes, tclObj being the primary one (tclVar and tkwin are two other important ones). The tclObj objects print with the leading <Tcl>. The string representation of objects of class tclObj is returned by tclvalue or by coercion through the as.character function. These two differ in how they treat spaces and new lines. Conversion to vectors of mode character, double, integer and logical is possible, though, in general, direct conversion of complicated Tcl expressions is not supported. One can create objects of this class through as.tclObj.

tcltk convenience functions The Tk extensions to Tcl have a complicated command structure, and thankfully, tcltk provides some more conveniently named functions. To illustrate, the Tcl command to configure the text property for a label object (.label) would look like

```
% .label configure -text "new text"
```

The tcltk package provides a corresponding function tkconfigure. The above would be done in an R-like way as (assuming lab is a label object):

```
tkconfigure(lab, 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

[3] The add-on package tcltk2 uses both techniques to enhance the base tcltk package with some open-source Tk extensions.

```
tcl(widget, subcommand, key=value, callback)
   /         |         |         \
widget$ID  subcommand   -key value   makeCallback
```

Figure 1.2: How the `tcl` function maps its arguments

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 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`. [4]

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 passes along strings as subcommands (`configure`);
- 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 allowing R functions and expressions to be called.

The `tcl` function uses position to create its command. The order of the subcommands needs to match that of the Tk API, so although it is true that often the R object is first, this is not always the case.

[4] The package `tcltk` was written before namespaces were implemented in R, so the "tk" prefix serves that role.

1.2 Constructors

In this Chapter, we will stick to a few basic widgets: labels, entry widgets, and buttons; 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 shell, wish, is used or the Tk package is loaded through the Tcl command `package require Tk`, a top level window named "." is created. (This is `.TkRoot`.) In the variable name `.label`, from above, the dot refers to the top level window. In `tcltk` a top-level window is created separately through the `tktoplevel` constructor, as was done in the example:

```
w <- tktoplevel()
```

Top-level windows will be explained in more detail in Chapter 2.

Other widget constructors require that a parent widget be specified. This is the first argument of the constructor. A typical invocation was given in the example.

```
l <- ttklabel(g, text="Enter your name:")
```

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> Enter your name:
```

```
tkcget(l, text=NULL)               # alternate syntax
```

```
<Tcl> Enter your name:
```

Coercion to character As mentioned, the `tclObj` objects can be coerced to characters in two ways. The conversion through `as.character` breaks the return value along whitespace:

```
as.character(tkcget(l, text=NULL))
```

```
[1] "Enter" "your"  "name:"
```

Whereas, conversion by the `tclvalue` function does not:

```
tclvalue(tkcget(l, text=NULL))
```

```
[1] "Enter your name:"
```

The `tkwidget` function

Constructors call the `tkwidget` function which returns an object of class `tkwin`. (In Tk the term “window” is used to refer to the drawn widget and not just a top-level window). E.g.,

```
str(btn)
```

```
List of 2
 $ ID : chr ".1.1.2.1"
 $ env:<environment: 0x100b77778>
 - 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 (**?**). As the construction of a new widget requires the `ID` and environment of its parent, the first argument to `tkwidget`, `parent`, must be a `tkwin` object, not simply its character ID, as is possible for the `tcl` function.

Geometry managers

In the example we saw several calls to `tkpack`. For example,

```
tkpack(f, expand=TRUE, fill="both")
tkpack(l, side="left")
tkpack(txt)
g1 <- ttkframe(f); tkpack(g1, anchor="ne")
```

As with Qt, when a new widget is constructed it is not automatically mapped. Tk uses geometry managers to specify how the widget will be drawn within the parent container. We will discuss two such geometry managers in Chapter 2 `tkpack` and `tkgrid`

The `tkack` command packs the widgets into the parent container in a box-like manner. The example shows various arguments that adjust the position of the child component and how space is to be allocated when an excess of space is present.

Tcl variables

For the button and label widgets in out example, their `text` property is configured through calls to their constructors. Many widgets allow an alternative way to specify one or two important properties using an independet Tcl variable.

In the call to `ttkentry` in the example we had:

```
txtVar <- tclVar("")
txt <- ttkentry(g, textvariable=txtVar)
```

The first line defines a new object of class `tclVar` which is used for the `textvariable` option when defining the entry widget. This variable is dynamically bound to the widget, so that changes to the variable are propagated to the GUI. (The Tcl variable is a model and the widget a view of the model.) The Tcl variable may be used with more than one widget, allowing a simple form of synchronization.

The basic functions involved are `tclVar` to create a Tcl variable, `tclvalue` to get the assigned value and `tclvalue<-` to modify the value.

```
tclvalue(txtVar) <- "Somebody's name"
tclvalue(txtVar)
```

```
[1] "Somebody's name"
```

Tcl variables have a unique identifier, returned by `as.character`:

```
as.character(txtVar)
```

```
[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 can't be recovered from the object itself. Hence, it must be found through R's scoping rules. (In Section 3.2 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
```

Commands

In the definition of the button we saw:

Figure 1.3: Comparison of themed versus non-themed dialog. The right one does not use an inner ttkframe and in addition to not having padding, has a mismatched color.

```
btn <- ttkbutton(g1, text="Click")
#
handler = function() print(sprintf("Hello %s", tclvalue(txtVar)))
tkconfigure(btn, command=handler)
```

Button widgets are used to allow direct access to some action, or command, and the `command` option is used to specify this. This may be given as a function or expression, though we only illustrate the former. The command is invoked by clicking and releasing the mouse on the button, by pressing the space bar when the button has the focus or by calling the widget's *ttkbutton* `invoke` subcommand.

The `command` option is available for many widgets, but is not the only means to invoke a function call, as Tk also allows one to bind to various types of events, e.g., button clicks. More on callbacks in `tcltk` will be explained in Section 1.3.

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.)

In our example, the toplevel window has a frame immediately packed inside of it through the commands:

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
```

The arguments to `tkpack` are given so that the frame, `f`, will expand and fill all the space allocated by the toplevel window. As the toplevel window is not a themed widget, not doing this can leave odd-looking effects.

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")
```

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(btn, "instate", "focus"))
```

```
[1] FALSE
```

To set a widget to be not sensitive to user input we have:

```
tcl(btn, "state", "disabled")           # not sensitive
```

```
<Tcl> !disabled
```

The states are bits and can be negated by prefacing the value with `!`:

```
tcl(btn, "state", "!disabled")          # sensitive again
```

```
<Tcl> disabled
```

The full list of states is in the manual page for `ttk::widget`.

The writing of themes will not be covered, but in Example 2.5 we show how to create a new style for a button. This topic is covered in some detail in the Tk tutorial by Roseman.

Window properties and state: `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.

To query the values of the containing window of the widget the `tkwinfo` function is used. When widgets are mapped, the "window" they are rendered to has properties, such as a class or size. There are a few subcommands provided by `tcltk`, but by no means is this exclusive. Rather, one can pass in the subcommand as an argument to `tkwinfo`. If the subcommand's 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 [5] 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> .4.1 .4.2
```

```
sapply(as.character(children), function(i) tkwinfo("class", i))
```

```
$`.4.1`
<Tcl> TButton

$`.4.2`
<Tcl> TButton
```

There are several possible subcommands, here we list a few. The *tkwinfo* `geometry` sub command returns the location and size of the 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.

Colors and fonts

Colors and fonts are typically specified through a theme, but at times it is desirable to customize the preset ones.

[5] 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.

Table 1.1: Standard font names defined by a theme.

Standard font name	Description
`TkDefaultFont`	Default font for all GUI items not otherwise specified
`TkTextFont`	Font for text widgets
`TkFixedFont`	Fixed-width font
`TkMenuFont`	Menu bar fonts
`TkHeadingFont`	Font for column headings
`TkCaptionFont`	Caption font (dialogs)
`TkSmallCaptionFont`	Smaller caption font
`TkIconFont`	Icon and text font

The 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 3.3 we show how to modify a style, as opposed to the `foreground` option, to change the text color in an entry widget.

Fonts Fonts are a bit more involved than colors. Tk version 8.5 made it more difficult to change font properties of individual widgets, 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 1.2. 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
```

Figure 1.4: A scrollable frame widget (cf. Example 3.11) showing the available fonts on a system.

```
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 1.4 shows a display of some available font families on a Mac OS X machine. See Example 3.11 for details.

The arguments for tkfont.create are optional. The size argument specifies the pixel size. The weight argument can be used to specify "bold" or "normal". Additionally, a slant argument can be used to specify either "roman" (normal) or "italic". Finally, underline and overstrike can be set with a TRUE or FALSE value.

Font metrics The average character size is important in setting the width and height of some components. (For example the text widget specifies its height in lines, not pixels.) These sizes can be found using the tkfont.measure and tkfont.metrics. Although the average text size varies for proportional fonts, the size of the M character is often used.

```
font_measure <- tcl("font","measure","TkTextFont","M")
fontWidth <- as.integer(tclvalue(font_measure))
tmp <- tkfont.metrics("TkTextFont","linespace"=NULL)
fontHeight <- as.numeric(tclvalue(tmp))
```

```
#
c(width=fontWidth, height=fontHeight)
```

```
width height
   10     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.) [6] 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`,

[6] The `tkrplot` package allows a third option `Rplot`. This package has the high-level command `tkrplot`, but the low-level use of a) calling `.my.tkdev(hscale=1,vscale=1)` b) creating a graphic and c) creating an image object through `tkimage.create("Rplot", img_name)` will produce a new image object one can use.

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)
```

1.3 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()
b <- ttkbutton(w, text="Some widget with the focus")
tkpack(b)
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

Of course, the possible events (or sequences of events) vary from widget to widget. In addition, these events can be specified in a few ways.

The example below uses two types of events. A single key press event, such as “C” or “O” can invoke a command and is specified by just its character. Whereas, the event of pressing the return key is specified using `Return`. In the following we bind the key presses to the top-level window and the return event to any button with the default class `TButton`.

Figure 1.5: Simple GUI showing buttons with `underline` property. The underlined letters match bindings to the top level window to invoke the button.

```
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 (Figure 1.5). 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 function 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 btn:

```
tkevent.generate(btn, "<<Save>>")
```

Example 1.1 uses virtual events to implement drag and drop features.

Callbacks

The tcltk package implements callbacks in a manner different from Tk, as the callback functions are R functions, not Tk procedures. This is much more convenient, but introduces some slight differences. In tcltk these callbacks can be expressions (unevaluated calls) or functions. We use only the latter. The basic callback function need not have any arguments and those that do 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 3.3) it can matter. [7]

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. [8]

% 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 1.1 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
  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:

[7]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.

[8]This event handling can prevent one being able to port some Tk code into `tcltk`. In those cases, one may consider sourcing in Tcl code directly.

```
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 1.1: Drag and Drop
This relatively involved example [9] shows several different uses of the event framework to implement drag and drop behavior between two widgets. It certainly may be skipped on first reading.

In `tcltk` much more work is involved with drag and drop, than with `RGtk2` and `qtbase`, as there are no predefined methods to facilitate the process.

Here we go through the steps needed to make one widget a drop source, and the other a drop target The basic idea is that when a value is being dragged, virtual events are generated for the widget the cursor is over. If that widget has callbacks listening to these events, then the drag and drop can be processed.

To begin, we create a simple GUI to hold three widgets. We use buttons for drag and drop, but only for convenience. Other widgets would be used in a real application.

```
w <- tktoplevel()
bDrag <- ttkbutton(w, text="Drag me")
bDrop <- ttkbutton(w, text="Drop here")
tkpack(bDrag)
tkpack(ttklabel(w, text="Drag over me"))
tkpack(bDrop)
```

[9] The idea for the example code originated with `http://wiki.tcl.tk/416`

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 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 <- ""
})
```

2

Tcl/Tk: Layout and Containers

2.1 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 3.5.)

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 1.2. 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 2.1). 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 1.3 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 3.3) 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 3.1.

```
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 2.1: 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)
}
```

2.2 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.

2.3 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 2.4 will illustrate that once again. When simultaneous horizontal and vertical alignment of child components is desired, the `tkgrid` function can be used to manage the components. [1]

A GUI may use a mix of `pack` and `grid` to 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. Unlike GTK+ or Qt, where boxes are packed in just one direction, these can be mixed and matched, but sticking to just one direction is typical, with nested frames to give additional flexibility.

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`. Example 3.8 shows a usage. 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")
```

[1] An excellent online reference, albeit for Perl/Tk, is *Learning Perl/Tk: Graphical User Interfaces with Perl* By Nancy Walsh. See `http://www.rigacci.org/docs/biblio/online/lperltk/ch02.html` for information about this topic.

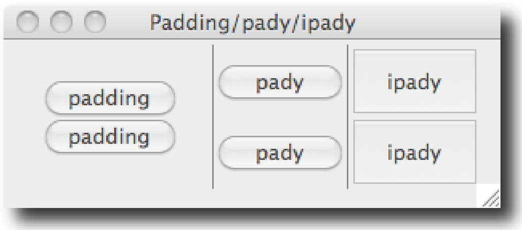

Figure 2.1: Various ways to put padding between widgets using `tkpack`. The `padding` option for the box container puts padding around the cavity for all the widgets. The `pady` option for `tkpack` puts padding around the top and bottom on the border of each widget. The `ipady` option for `tkpack` puts padding within the top and bottom of the border for each child (breaking the theme under Mac OS X).

```
#
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"))
#
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 2.1 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, tkpack.configure,  padx=10, pady=5)
```

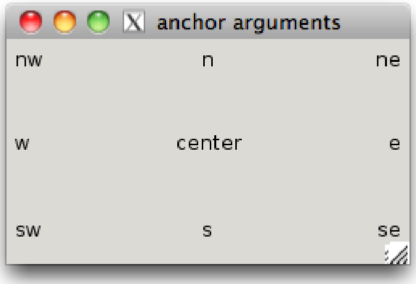

Figure 2.2: The `anchor` argument is specified through compass directions

Cavity model The packing algorithm, as described in the Tk documentation, is based on arranging where to place a slave into the rectangular unallocated space called a "cavity." We use the nicer terms "child component" and "box" to describe 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.

The anchor, expand, fill arguments When there is more space in the box than requested by the child component, there are other options. The `anchor` option can be used to anchor the child to a place in the box by specifying one of the valid compass points (eg. `"n"` or `"se"`) leaving blank space around the child (Figure 2.2.)

An alternative is to have one or more of the widgets expand to fill the available space. Each child packed in with the option `expand` set to `TRUE` will have the extra space allocated to it in an even manner. The `fill` option is used to base the size of the child on the available cavity in the box – not on the requested size of the child. The `fill` option can be `"x"`, `"y"` or `"both"`. The first two expanding the child's size in just one direction, the latter in both.

Example 2.2: Expand/fill options for `tkpack`

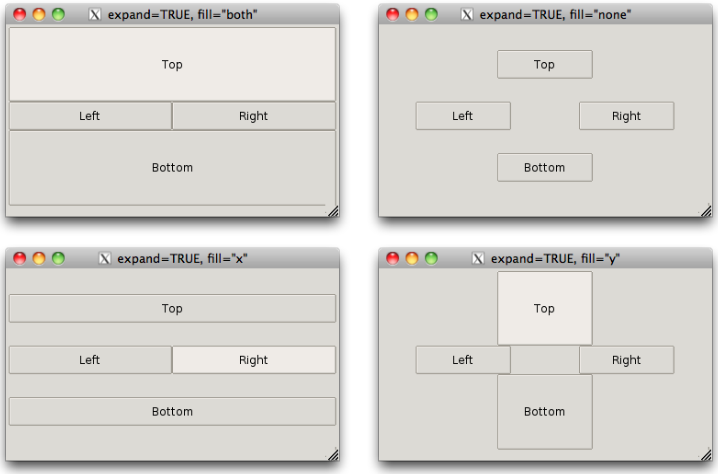

Figure 2.3: Similar layout with `expand=TRUE` but different values of `fill`. The space allocated to the top and bottom buttons through expansion fills the vertical area, as these were added with `side="top"` and `side="bottom"`; whereas the left and right buttons expand in the horizontal direction, as they were added with sides `left` and `right`. The different `fill` values direct the buttons to take up this allocated space in different manners.

Figure 2.3 shows examples of different values for `"fill"` when `expand=TRUE` is specified. We used the following code to create the images: [2]

```
w <- tktoplevel()
tkwm.title(w, "Expand/Fill arguments")
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
##
packButton <- function(txt, ...)
  tkpack(b <- ttkbutton(f, text=txt))
##
packButton("Top",    side="top",    expand=TRUE, fill="both")
packButton("Bottom", side="bottom", expand=TRUE, fill="both")
packButton("Left",   side="left",   expand=TRUE, fill="both")
packButton("Right",  side="right",  expand=TRUE, fill="both")
```

Modifying the fill styles was easy, for example

```
children <- as.character(tkwinfo("children", f))
sapply(children, tkpack.configure, fill="none")
```

[2] This example follows one of (?)

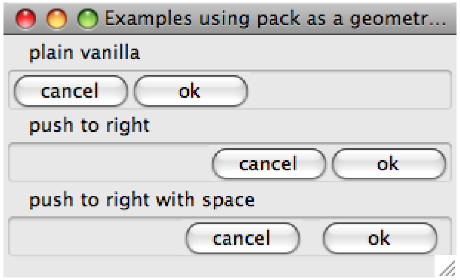

Figure 2.4: Demonstration of using `tkpack` options showing effects of using the `side` and `padx` options to create dialog buttons.

Not enough space When the toplevel window does not have sufficient space to satisfy the combined size requests of its child components either some widgets will be covered or one can resize the toplevel window. When components are covered, the ones that are packed in first are given highest priority in the size request.

To force a recompuation of of the size of the toplevel window, one can call the *wm* `geometry` subcommand with an empty string:

```
tkwm.geometry(tt, "")
```

The toplevel window, `tt` above, can be recovered from a child component, say `b`, through

```
tkwinfo("toplevel", b)
```

propagate In Example 3.6 we define a convenience function for creating a table widget. There we have a call to the subcommand *pack* `propagate`. This prevents the querying of the child widgets to compute the size request. In the example, this is useful as the scrollbars used should depend on the size requested by the parent, and not the underlying table widget.

Example 2.3: Packing dialog buttons

This example shows how one can pack in action buttons, such as when a dialog is created.

The first example just uses `tkpack` without any arguments except the side to indicate 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), tkpack, side="left")
```

Figure 2.5: Example of a simple dialog

Typically the buttons are right justified. One way to do this is to pack in using `side` with a value of `"right"`. This shows how to use a blank expanding label to take up the space on the left.

```
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), tkpack, 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), tkpack, side="left", padx=6)
```

Example 2.4: A non-modal dialog

This example shows how to use a window, frames, labels, buttons, icons, packing and bindings to create a non-modal dialog.

Although 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` 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:

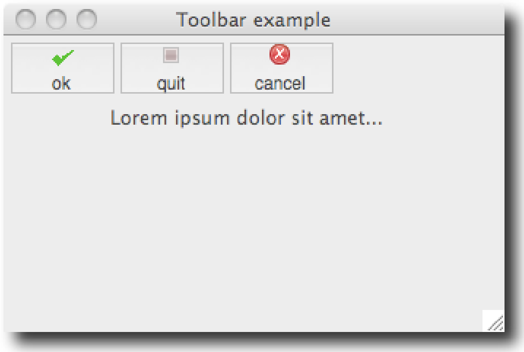

Figure 2.6: Illustration of using `tkpack` and `tkgrid` to make a toolbar.

```
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 2.5: Using `tkgrid` to create a toolbar

Tk does not have a toolbar widget. Here we use `tkgrid` to show how we can add one to a top-level window in a manner that is not affected by resizing. We begin by packing a frame into a top-level window.

```
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 2.6: 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),
           column=day.of.week(year, month, day),
           sticky="e")
  })
}
```

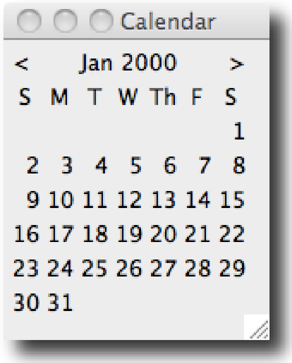

Figure 2.7: A monthly calendar illustrating various layouts.

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))
})
```

2.4 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:

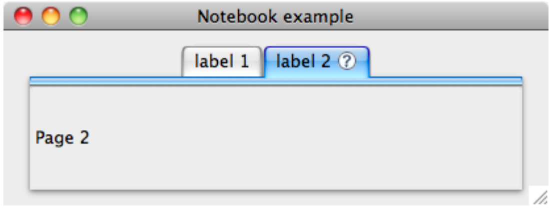

Figure 2.8: A basic notebook under Mac OS X

```
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> 54
```

Notebooks

Tabbed notebook containers are produced by the `ttknotebook` constructor. Notebook pages can be added through the *ttknotebook* `add` subcommand or inserted after a page through the *ttknotebook* `insert` subcommand. The latter requires a tab ID to be specified, as described below. Typically, the child components would be containers to hold more complicated layouts. The tab label is configured similarly to `ttklabel` through the options `text` and (the optional) `image`, which if given has its placement determined by `compound`. The placement of the child component within the notebook page is manipulated similarly as `tkgrid` through the `sticky` option with values specified through compass points. Extra padding around the child can be added with the `padding` option.

Tab identifiers Many of the commands for a notebook require a specification of a desired tab. This can be given by index, starting at 0; by the values `"current"` or `"end"`; by the child object added to the tab, 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 2.8):

```
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.

3

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.

3.1 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:

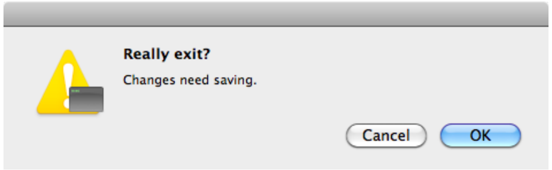

Figure 3.1: A basic modal dialog constructed by `tkmessageBox`.

```
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)
```

There a different ways to use `tkwait`. The function `tkwait.window` will make a toplevel window modal waiting until it is destroyed. In the following we 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 `tkgetOpen-File`, `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 3.1: A "File" menu
To illustrate, a simple example for a file menu (Section 3.5) 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="#ababab")
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)
})
```

3.2 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

Figure 3.2: A dialog to collect values for a *t* test described in Example 3.2 showing several of the selection widgets discussed in the section: a check button, radio button, combo boxes, a scale widget and a spinbox.

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> ::RTcl5
```

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 3.2 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 in general 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 3.3.

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", ...)
```

[1] As of the version of `tcltk` accompanying R 2.13.1

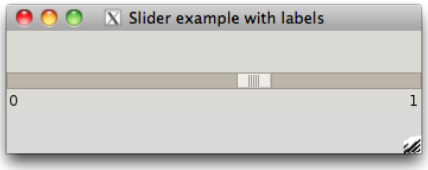

Figure 3.3: The `ttk::scale` widget with labels added

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 (Figure 3.2). We write this using an R reference class.

```
Slider <-
  setRefClass("TtkSlider",
     fields=c("frame", "widget", "v", "x", "FUN"),
     methods=list(
       initialize=function(parent, x) {
         x <<- x;  v <<- tclVar(1)
         FUN <<- NULL                   # NULL of fuction
         frame <<- ttkframe(parent)
         widget <<- ttkscale(frame, from=1, to=length(x),
                             variable=v, orient="horizontal")
         ## For this widget, the callback is passed a value
         ## which we ignore here
         tkconfigure(widget, command=function(...) {
           if(is.function(FUN)) FUN(.self)
         })
         layout_gui()
         .self
       },
       layout_gui=function() {
         tkgrid(widget, row=0, column=0, columnspan=3,
                sticky="we")
         tkgrid(ttklabel(frame, text=x[1]),
                row=1, column=0)
         tkgrid(ttklabel(frame, text=x[length(x)]),
                row=1, column=2)
         tkgrid.columnconfigure(frame, 1, weight=1)
```

```
        },
        add_callback=function(FUN) FUN <<- FUN,
        get_value=function() x[as.numeric(tclvalue(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()
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
x <- seq(0,1,by=0.05)
##
s <- Slider$new(parent=w, x=x)
tkpack(s$frame, expand=TRUE, fill="x", anchor="n")
##
s$set_value(0.5)
print(s$get_value())
```

```
[1] 0.5
```

As seen in the `initialize` and `get_value` methods, the `variable` option can be used for specifying a Tcl variable to record the value of the slider. This is convenient when the variable and widget are encapsulated into a class, as above. Otherwise the `value` option is available. The `tkget` and `tkset` 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.

The `add_callback` method can be used to add a callback function when the slider changes value. We set up the call to pass back in a reference to the object, so there is no issue with finding the Tcl variable to get the value.

```
s$add_callback(function(obj) print(obj$get_value()))
```

Spin boxes

A themeable spinbox is introduced in Tk version 8.5.9. However, as of writing, the Window libraries accompanying R are 8.5.8, so we will assume 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: [2]

```
tkspinbox <- function(parent, ...)
    tkwidget(parent, "tk::spinbox", ...)
```

The non-themeable widgets have many more options than the themeable ones, as style properties can be set on a per-widget basis. We won't discuss those here. The 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 3.3.

Example 3.2: 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 3.3.

[2] One could compare the result of `tcl("info", "patchlevel")` to 8.5.9 and use `"ttk::spinbox"` if the libraries support it.

We will use a data store to hold the values to be passed to `t.test`. For the data store, we use an environment to hold Tcl variables.

```
### Data model
e <- new.env()
e$x <- tclVar(""); e$f <- tclVar(""); e$data <- tclVar("")
e$mu <- tclVar(0); e$alternative <- tclVar("two.sided")
e$conf.level <- tclVar(95); e$var.equal <- tclVar(FALSE)
```

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)
```

The grid will have four columns, with columns 0 and 2 being for labels. We don't want the labels to expand the same way we want the widget columns to 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)
```

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 use a `ttkentry` widget (Section 3.3) 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$conf.level)
confSpin <- tkspinbox(confFrame, from=75, to=100, increment=1,
                     textvariable=e$conf.level, width=5)
##
tkpack(confScale, expand=TRUE, fill="y", side="left")
tkpack(confSpin, side="left")
```

A checkbox is used to collect the logical value 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)
```

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
sapply(list(cancel, ok), tkpack, side="left", padx=6)
```

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())
```

3.3 Text widgets

Tk provides both single- and multi-line text entry widgets. The section describes both.

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 3.3: Putting in an initial message
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 Tcl variable and the initial message. The main properties we have are defined via

```
setOldClass(c("tkwin", "tclVar"))
TtkEntry <- setRefClass("TtkEntry",
                        fields=list(
                          e="tkwin",    # entry
                          v="tclVar",   # textvariable
                          init_msg="character"
                          ))
```

We need to indicate to the user that the initial message is not the current text. We do so with a style. It simply sets the foreground (text) color to gray.

```
.Tcl("ttk::style configure Gray.TEntry -foreground gray")
```

Now we create methods to accomodate the initial message. We have methods `is_init_msg`, to compare the current text with the intial message; and `show_init_msg` and `hide_init_msg` to toggle the messages. The only novelty is using the `style` option for a themeable widget.

```
TtkEntry$methods(
                 is_init_msg = function() {
                   "Is the init text showing?"
                   as.character(tclvalue(v)) == init_msg
                 },
                 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)
                 })
```

Our accessor methods, set_text and get_text, must work around a possible intial message.

```
TtkEntry$methods(
                 set_text=function(text, hide=TRUE) {
                   "Set text into widget"
                   if(hide) hide_init_msg()
                   v_local <- v         # avoid warning
                   tclvalue(v_local) <- text
                 },
                 get_text=function() {
                   "Get the text value"
                   if(!is_init_msg())
                     as.character(tclvalue(v))
                   else
                     ""
                 })
```

In the initialize method, we will add bindings to switch between the initial message and the entry area. We use the focus in and out events to initiate this.

```
TtkEntry$methods(
                 add_bindings=function() {
                   "Add focus bindings to make this work"
                   tkbind(e, "<FocusIn>", hide_init_msg)
                   tkbind(e, "<FocusOut>", function() {
                     if(nchar(get_text()) == 0)
                       show_init_msg()
                   })
                 })
```

Finally, our initialization method follows.

```
TtkEntry$methods(
         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
         })
```

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 e$e.

```
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 3.4: 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`, `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)
b <- ttkbutton(f,text="click")        # something to focus on
sapply(list(e, b), tkpack, side="left", padx=2)
```

```
[[1]]

[[2]]
```

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.

```
xscr <- ttkscrollbar(parent, orient="horizontal",
                  command=function(...) tkxview(widget, ...))
yscr <- ttkscrollbar(parent, orient="vertical",
                  command=function(...) tkyview(widget, ...))
```

The `tkxview` and `tkyview` functions 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 are no simple means to hide scrollbars when not needed, although the `tcltk2` package 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.

```
w <- tktoplevel()
tkwm.title(w, "Simple tktext example")
txt <- tktext(w)
addScrollbars(w, txt)
```

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 1.2). 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).

Getting text The *tktext* `get` subcommand is used to retrieve the text in the buffer. One specifies what part of the text buffer should be returned using indices (below). The following shows how to retrieve the entire contents of the buffer:

```
value <- tkget(txt, "1.0", "end")
as.character(value)                    # wrong way
```

```
character(0)
```

```
tclvalue(value)
```

```
[1] "\n"
```

The return value is of class `tclObj`. Coercion to character should be done with `tclvalue` and not `as.character` to preserve the distinction between spaces and line breaks.

Indices As with the entry widget, several commands take indices to specify position within the text buffer. Only for the 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.)

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(txt, "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(txt, "1.0", "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(txt, "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(txt,"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(txt, "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", txt)
```

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(txt, "leftlimit", "insert")
tkmark.gravity(txt, "leftlimit", "left")    # keep onleft
tkinsert(txt, "insert", "new text")
tkget(txt, "leftlimit", "insert")
```

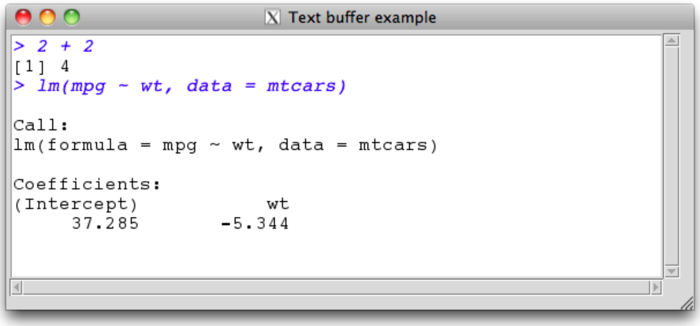

Figure 3.4: A text widget used to show formatted R commands and their output

```
<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(txt, "edit", "undo")                    # no output
tcl(txt, "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 3.5: 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 envision this as a piece of a larger GUI which generates the commands to evaluate. For this example though, we make a simple GUI (Figure **??**).

```
w <- tktoplevel(); tkwm.title(w, "Text buffer example")
f <- ttkframe(w, padding=c(3,3,12,12))
```

```
tkpack(f, expand=TRUE, fill="both")
txt <- tktext(f, width=80, height = 24)   # default size
addScrollbars(f, txt)
```

To distinguish between commands and their output we define the following tags:

```
tktag.configure(txt, "commandTag", foreground="blue",
                font="courier 12 italic")
tktag.configure(txt, "outputTag", font="courier 12")
tktag.configure(txt, "errorTag", foreground="red",
                font="courier 12 bold")
```

The following function does the work of evaluating a command chunk then inserting the values into the text buffer, using the different markup tags specified above to indicate commands from output.

```
evalCmdChunk <- function(txt, 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(txt, "end", command, "commandTag")
    tkinsert(txt, "end","\n")
    ## output, should check for errors in eval!
    output <- capture.output(eval(cmd, envir=.GlobalEnv))
    output <- paste(output, collapse="\n")
    tkinsert(txt, "end", output, "outputTag")
    tkinsert(txt, "end","\n")
  }
}
```

This is how it can be used.

```
evalCmdChunk(txt, "2 + 2; lm(mpg ~ wt, data=mtcars)")
```

3.4 Treeview widget

The themed treeview widget can be used to display rectangular data, like a data frame; or hierarchical data, like a list. The usage is similar, but for a minor change to indicate the hierarchical structure.

Rectangular data

The `ttktreeview` constructor creates the tree widget. There is no separate model for this widget, as there is in GTK+ or Qt, but there is a means to adjust what is displayed. The argument `columns` is used to specify internal names for the columns and indicate the number of columns. A value of `1:n` will work here unless explicit names are desired. The argument `displaycolumns` 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 requested 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 treeview widget with just one column showing 25 rows at a time, 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)                 # our 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 the root `""` 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 seq_along(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. [3] 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 allowed the use of the subcommand *ttktreeview* tag configure to configure the background color. In addition, one can adjust foreground color, font or image for an item.

Column properties The columns can be configured on a per-column basis. Columns can be referred to by the name specified through the columns argument or by number starting at 1 with "#0" referring to the tree column. The column headings can be set through the *ttktreeview* heading subcommand. The heading, similar to the button widget, can be text, an image or both. The text placement of the heading may be positioned through the anchor option. For example, this command will center the text heading of the first column:

```
tcl(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

[3] As does wrapping the values within braces.

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")
```

Example 3.6: A convenience function for populating a table
We put the above commands together into a convenience function for subsequent use. The following assumes m is a character matrix. It returns a list containing the enclosing frame and the treeview object.

```
populate_rectangular_treeview <- function(parent, m) {
  enc_frame <- ttkframe(parent)
  frame <- ttkframe(enc_frame)
  tkpack(frame, expand=TRUE, fill="both")
  tr <- ttktreeview(frame,
                    columns=seq_len(ncol(m)),
                    show="headings")
  addScrollbars(frame, tr)
  tkpack.propagate(enc_frame, FALSE)    # size from frame
  ## headings,widths
  font_measure <- tcl("font","measure","TkTextFont","0")
  charWidth <- as.integer(tclvalue(font_measure))
  sapply(seq_len(ncol(m)), function(i) {
    tcl(tr, "heading", i, text=colnames(m)[i])
    tcl(tr, "column", i,
        width=10 + charWidth*max(apply(m, 2, nchar)))
  })
  tcl(tr, "column", ncol(m), stretch=TRUE)
  ## values
  if(ncol(m) == 1)  m <- as.matrix(paste("{", m , "}", sep=""))
  apply(m, 1, function(vals)
    tcl(tr, "insert", "", "end", values=vals))
  ##
  return(list(tr=tr, frame=enc_frame))
}
```

The use of tkpack.propagate allows us to control the size of the enclosing component by configuring the size of the enclosing frame. Otherwise, in the computation for requested size, the treeview widget will respond with a width computed by its column widths. However, we use a horizontal scrollbar to avoid this.

To use this we need to configure the size of the scrollable frame widget. For example:

```
w <- tktoplevel()
```

```
m <- sapply(mtcars, as.character)
a <- populate_rectangular_treeview(w, m)
tkconfigure(a$tr, selectmode="extended") # multiple selection
tkconfigure(a$frame, width=300, height=200) # frame size
tkpack(a$frame, expand=TRUE, fill="both")
```

Item IDs Each row has a unique item ID generated by the widget when a row is added. The base ID is `""` (why this 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] "Universidad Nacional de La Plata"
```

The value returned from the `item` command can be difficult to parse, as Tcl places braces around values with blank spaces. The coercion through `as.character` works much better at extracting the individual columns. A possible alternative to using the `item` command, is to instead keep the original data frame and use the index of the item to extract the value from the original. Since the data from the widget is character data, this can be much preferred to having to coerce values to their original class.

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 3.7: 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.

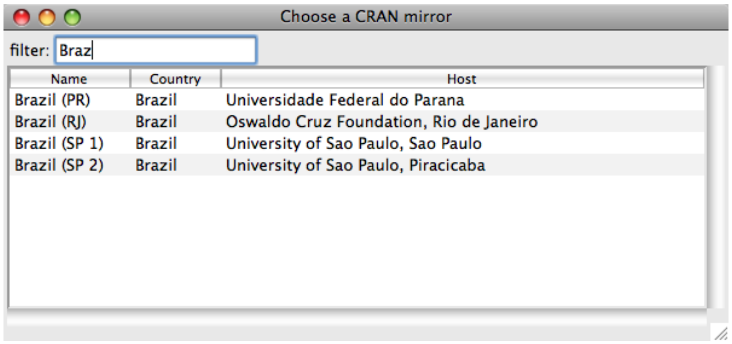

Figure 3.5: Using `ttktreeview` to show various CRAN sites. This illustration adds a search-like box to filter what repositories are displayed for selection.

```
f0 <- ttkframe(f); tkpack(f0, fill="x")
l <- ttklabel(f0, text="filter:"); tkpack(l, side="left")
filterVar <- tclVar("")
filterEntry <- ttkentry(f0, textvariable=filterVar)
tkpack(filterEntry, side="left")
```

The treeview will only show the first three columns of the data frame, although we store the fourth which contains the URL.

```
f1 <- ttkframe(f); tkpack(f1, expand=TRUE, fill="both")
tr <- ttktreeview(f1, columns=1:ncol(df),
                  displaycolumns = 1:(ncol(df) - 1),
                  show = "headings",     # not "tree"
                  selectmode = "browse") # single selection
addScrollbars(f1, tr)
```

We configure the column widths and titles as follows:

```
widths <- c(100, 75, 400)            # hard coded
nms <- names(df)
for(i in 1:3) {
  tcl(tr, "heading", i, text=nms[i])
  tcl(tr, "column", i, width=widths[i],
      stretch=TRUE, anchor="w")
}
```

The treeview widget does not do filtering internally. [4] 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) {
```

[4]The model-view-controller architecture of GTK+, say, makes this task much easier, as it allows for an intermediate proxy model.

```
  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")
}
```

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))
})
```

Example 3.8: A dialog for subsetting a data frame

This longish example creates a framework for showing a list of similar items, whose length is uncertain. There are several uses of such a framework. For example, a GUI for formulas might have items given by terms between + values, or a GUI for `ggplot2` might have items which represent individual layers of a plot. Here we use the framework to create

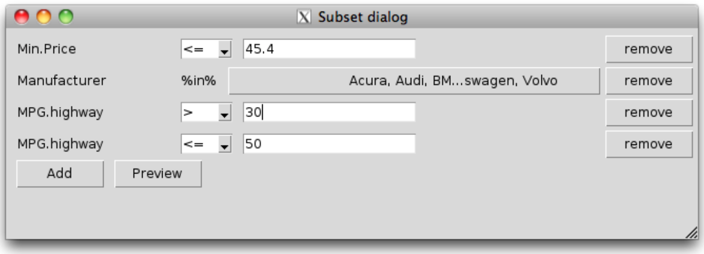

Figure 3.6: A dialog for subsetting a data frame

a dialog for the `subset` argument of the `subset` function. [5] That argument combines an arbitrary number of statements that produce logical values to produce a logical index for a data frame. For our framework, each item will produce one of these logical statements, and our list will hold the items.

To implement this, we first create a `FilterList` class. Our class has a few properties: `df` to hold the data frame; `l` to hold the list items; `id` to hold an internal counter to reference the list items by; and `frame` to hold a `ttkframe` instance, the parent container for each item.

```
setOldClass(c("tkwin","tclVar"))
FilterList <- setRefClass("FilterList",
                          fields=list(
                            df="data.frame",
                            l="list",
                            id="integer",
                            frame="tkwin"
                            ))
```

The main interface for a filter list is limited. For management, we define a method to `add` a list item and one to `remove` a list item. We also need a method (`get_value`) to analyze the items and produce a logical vector for subsetting the data frame with. Beyond that we have methods to setup the GUI, a `preview` method to see the current subsetting, and a method to select a variable from the data frame.

First, we define a method to setup our GUI. The `initialize` method will be passed a parent container. Here we pack in a frame (`enc_frame`) to hold the pieces of our GUI. [6] These consist of a frame to hold the items and a frame to hold the buttons. We use the `tkgrid` layout manager which allows us to grow the top frame as needed, yet have the buttons receive the additional expanding space.

[5] The author's would like to thank Liviu Andronic ideas related to this example.

[6] This means that `tkpack` needs to be used to manage this parent. An alternative would be to pass back the enclosing frame object so that it can be managed as the user wants.

```
FilterList$methods(
          setup_gui = function(parent) {
            enc_frame <- ttkframe(parent, padding=5)
            tkpack(enc_frame, expand=TRUE, fill="both")
            frame <<- ttkframe(enc_frame)
            button_frame <- ttkframe(enc_frame)
            ## use grid to manage these
            tkgrid(frame, sticky="news")
            tkgrid(button_frame, sticky="new")
            tkgrid.rowconfigure(enc_frame, 1, weight=1)
            tkgrid.columnconfigure(enc_frame, 0, weight=1)
            ##
            addBtn <-
              ttkbutton(button_frame, text="Add",
                        command=function() .self$add())
            previewBtn <-
              ttkbutton(button_frame, text="Preview",
                        command=function() .self$preview())
            ##
            sapply(list(addBtn, previewBtn), tkpack,
                   side="left", padx=5)
          })
```

The initialize method simply initializes our fields and then sets up the GUI. As the point of this is to filter a data frame, the df argument has no default value and must be specified.

```
FilterList$methods(
           initialize=function(df, parent, ...) {
             initFields(df=df, l=list(), id=0)
             setup_gui(parent)
             callSuper(...)
           })
```

Before showing a filter, we force the user to select a variable to filter by. This selection involves choosing one from possibly many. A table is an excellent choice for this, as it gracefully handles many values. This convenience method provides a table selection widget in a modal dialog window. Selection happens when a user selects one of the rows of the table.

```
FilterList$methods(
           select_variable=function() {
             "Return a variable name from the data frame"
             x <- sapply(df, function(i) class(i)[1])
             m <- cbind(Variables=names(x), Type=x)
             w <- tktoplevel()
             f <- ttkframe(w, padding=c(3,3,3,12))
             tkpack(f, expand=TRUE, fill="both")
```

```
              ##
              a <- populate_rectangular_treeview(f, m)
              tkconfigure(a$frame, width=300, height=200)
              tkpack(a$frame, expand=TRUE, fill="both")
              ## select a value, store in out
              out <- NA
              tkbind(a$tr, "<<TreeviewSelect>>", function(W) {
                sel <- tcl(W, "selection")
                val <- tcl(W, "item", sel, "-values")
                assign("out", as.character(val)[1], inherits=TRUE)
                tkdestroy(w)
              })
              tkwait.window(w)
              return(out)
            })
```

Our main add method has a few tasks: to select a variable, to create a new filter item, create a container, do the internal bookkeeping, and finally to call the items `make_gui` method. The `newFilterItem` call is an S3 generic, whose dispatch finds the correct filter item class.

```
FilterList$methods(
            add=function(varname, ...) {
              if(missing(varname))
                varname <- select_variable()
              x <- get(varname, df)
              ## new item
              id <<- id + 1
              item <- newFilterItem(x, varname, id, .self)
              ## make frame
              enc_frame <- ttkframe(frame)
              tkpack(enc_frame, expand=TRUE, fill="both", pady=2)
              l[[as.character(id)]] <<- list(frame=enc_frame,
                                             item=item)
              item$make_gui(enc_frame)
            })
```

To remove an object requires us to remove it from our internal list and from the GUI. We use `tkpack` to manage the items, so `tkpack.forget` is used to remove the item. In the `add` method we store the enclosing frame to make this task easy.

```
FilterList$methods(
            remove=function(id_obj, ...) {
              "Remove. id is character or item object"
              if(!is.character(id_obj))
                id_obj <- id_obj$id
              tkpack.forget(l[[id_obj]]$frame)
              l[[id_obj]] <<- NULL
```

```
            })
```

Here we query all the items and combine them to create a logical index vector. The item interface described below will provide its own `get_value` method so this task is a matter of combining the results of each of those calls. We use `all` here, but if one wanted to extend this GUI, one area would be to allow the user to specify “and” or “or” between each item.

```
FilterList$methods(
            get_value = function() {
              "Return logical value for all filter items"
              if(length(l) == 0)
                return(rep(TRUE, length=nrow(df)))
              ##
              out <- sapply(l, function(i) i$item$get_value())
              out[is.na(out)] <- FALSE   ## coerce NA to FALSE
              apply(out, 1, "all")
            })
```

The `get_value` method makes it easy to provide a `preview` method to show the current state of the subsetting. Basically we just need to create a character matrix that we want to display and then use our previously defined `populate_rectangular_treeview` function.

```
FilterList$methods(
            preview = function() {
              "Preview data frame"
              ind <- get_value()
              if(!any(ind)) {
                message("No matches")
                return()
              }
              ## coerce to character
              m <- fl$df[ind,]
              for(i in seq_along(m)) m[,i] <- as.character(m[,i])
              ##
              w <- tktoplevel()
              f <- ttkframe(w, padding=c(3,3,3,12))
              tkpack(f, expand=TRUE, fill="both")
              a <- populate_rectangular_treeview(f, m)
              tkconfigure(a$frame, width=400, height=300)
              tkpack(a$frame, expand=TRUE, fill="both")
              ##
              btn <- ttkbutton(f, text="dismiss",
                               command=function() tkdestroy(w))
              tkpack(btn, anchor="sw")
              tkwait.window(w)
            })
```

To use this new class, we would integrate it into a dialog, the basic call needed would be something along the lines of the following:

```
w <- tktoplevel()
require(MASS)
flist <- FilterList$new(df=Cars93, parent=w)
```

But before that will work, we need to define the filter item classes.

Filter items As mentioned, we use an S3 generic to select the appropriate filter item. These are still be be defined, but we show the default choice.

```
newFilterItem <- function(x, nm=deparse(substitute(x)), id,
                          list_ref) UseMethod("newFilterItem")
newFilterItem.default <- function(x, nm=deparse(substitute(x)),
                                  id, list_ref)
  FilterItemNumeric$new(x=x, nm=nm, id=id, list_ref=list_ref)
```

A filter item needs to produce a logical vector used for indexing. At a minimum we require a few properties: `x` to store the variable's data that we are considering; `nm` to store the name of this variable; `id` to store the id of where this item is stored in the filter list; and `list_ref` to store a reference to the filter list.

```
FilterItem <- setRefClass("FilterItem",
                          fields=list(
                            x="ANY",
                            nm = "character",
                            id = "character",
                            list_ref="ANY"
                            ))
```

The filter item interface is not complicated. The most important method is `get_value` to return a logical variable. This was called by the filter list's similarly named `get_value` method. As well, we call the item's `make_gui` method in the filter list. The last method is simply a `remove` method which calls back up into `remove` method of the item's parent filter list.

```
FilterItem$methods(
           initialize=function(...) {
             initFields(...)
             .self
           },
           get_value = function() {
             "Return logical value of length x"
             stop("Must be subclassed")
           },
           remove=function() list_ref$remove(.self),
           make_gui = function(parent, ...) {
             "Set up GUI, including defining widgets"
```

```
              removeBtn <- ttkbutton(parent, text="remove",
                                     command=function() {
                                       .self$remove()
                                     })
              tkpack(removeBtn, side="right")
            })
```

The interesting things happen in the subclasses. For numeric values we add two new properties to help with our `get_value` method: one to store an inequality operator and one to store an expression the user can enter.

```
FilterItemNumeric <- setRefClass("FilterItemNumeric",
                                 contains="FilterItem",
                                 fields=list(
                                   ineqVar="tclVar",
                                   valVar="tclVar"
                                   ))
```

With these two properties, our `get_value` method becomes a matter of pasting together an expression then evaluating it. We evaluate this within the data frame so that `valVar` could use variable names from the data framed.

```
FilterItemNumeric$methods(
          get_value = function() {
            xpr <- paste(nm, tclvalue(ineqVar), tclvalue(valVar))
            eval(parse(text=xpr), envir=list_ref$df, parent.frame())
          })
```

Our GUI has three widgets a label, a combobox for the inequality and an entry widget to put in values. One could simplify this, say wth a slider to slide through the possible values, but using an entry widget gives more flexibility in the specification. We see that we simply pack these widgets into the parent that is passed in to the method call.

```
FilterItemNumeric$methods(
          make_gui = function(parent) {
            ## standard width for label
            labWidth <- max(sapply(names(list_ref$df), nchar))
            lab <- ttklabel(parent, text=nm, width=labWidth)
            ## ineq combo
            vals <- c(">=", ">", "==", "!=", "<", "<=")
            ineqVar <<- tclVar("<=")
            ineq <- ttkcombobox(parent, values=vals,
                                textvariable=ineqVar, width=4)
            ## entry
            valVar <<- tclVar(max(x, na.rm=TRUE))
            val <- ttkentry(parent, textvariable=valVar)
```

```
          ##
          sapply(list(lab, ineq, val), tkpack, side="left",
                 padx=5)
          callSuper(parent)
        })
```

The character selection class, also used with factors, is more involved. Our `get_value` method is basically `x %in% cur_vals`, where `cur_vals` is a selection from all possible values. We might want to use a group of checkboxes here, but that can get unwieldly when there are more than a handful of choices. [7] We opt instead for a table selection widget. That can take up vertical screen space. To avoid this we use a button which shows the currently selected values, that can be clicked to open a dialog to adjust these values. To keep a consistent horizontal size to these buttons we "ellipsize" the button's text in the `ellipsize` method. Some graphical toolkits, but not Tk, have built-in "ellipsize" methods which prove useful when controlling space allocations when translations are involved, as these can vary widely in the number of characters needed to display.

For our new subclass, we have four additional properties, the tree view for selection, the button, and vectors to store the possible values and the currently selected values.

```
FilterItemCharacter <-
  setRefClass("FilterItemCharacter",
              contains="FilterItem",
              fields=list(
                tr="tkwin",
                btn="tkwin",
                poss_vals="character",
                cur_vals="character"
                ))
```

As mentioned, our `get_value` method is easy to define:

```
FilterItemCharacter$methods(
        get_value = function() {
          x %in% cur_vals
        })
```

The main work is in our `select_values_dialog`, defined below. We use the following helper function to preselect the currently selected values when the dialog is opened.

```
sel_by_name <- function(tr, nms) {
  all_ind <- as.character(tcl(tr, "children", ""))
```

[7] A table of checkboxes might also be used, but this isn't directly supported by the treeview widget of `tcltk`. Although, for the intrepid, one could set the image attribute for each row to show a check or non-check depending on the state.

```
  vals <- sapply(all_ind, function(i) {
    as.character(tcl(tr, "item", i, "-values"))
  })
  ind <- names(vals[vals %in% nms])
  sapply(ind, function(i) tcl(tr, "selection", "add", i))
  sapply(setdiff(all_ind, ind),
         function(i) tcl(tr, "selection", "remove", i))
}
```

This is the main dialog to select values. Here multiple selection is achieved by extending the selection through holding the shift and control keys while clicking on items.

```
FilterItemCharacter$methods(
          select_values_dialog=function() {
            w <- tktoplevel()
            f <- ttkframe(w, padding=c(3,3,12,12))
            tkpack(f, expand=TRUE, fill="both")
            tkpack(ttklabel(f,
              text="Select your values by extending selection"))
            ## selection
            m <- matrix(poss_vals); colnames(m) = "Values"
            a <- populate_rectangular_treeview(f, m)
            tkconfigure(a$tr, selectmode="extended")
            tkconfigure(a$frame, width=200, height=300)
            tkpack(a$frame, expand=TRUE, fill="both")

            sel_by_name(a$tr, cur_vals)         # see above

            tkbind(a$tr, "<<TreeviewSelect>>", function() {
              ind <- as.character(tcl(a$tr, "selection"))
              cur <- sapply(ind, function(i) {
                as.character(tcl(a$tr, "item", i, "-values"))
              })
              if(length(cur) == 0)
                cur <- character(0)
              cur_vals <<- cur
            })
            ## buttons
            f1 <- ttkframe(f); tkpack(f1)
            toggleBtn <- ttkbutton(f1, text="toggle",
                          command=function() toggle_sel(a$tr))
            setBtn <- ttkbutton(f1, text="set",
                          command=function() tkdestroy(w))
            sapply(list(toggleBtn, setBtn), tkpack,
                   side="left", padx=5)
            ## make modal
            tkwait.window(w)
```

```
            tkconfigure(btn, text=ellipsize())
          })
```

Here is our previously mentioned convenience method to make the button size uniform by "ellipsizing" the button's label.

```
FilterItem$methods(ellipsize = function() {
            tmp <- paste(cur_vals, collapse=", ")
            if((N <- nchar(tmp)) > 50)
              tmp <- sprintf("%s...%s", substr(tmp, 0, 15),
                             substr(tmp, N-12,N))
            sprintf("%50s", tmp)
          })
```

Our main GUI for a character or factor item then has three widgets: labels for the name and `%in%` operator and a button.

```
FilterItemCharacter$methods(make_gui = function(parent) {
            poss_vals <<- sort(unique(as.character(x)))
            cur_vals <<- poss_vals
            ## label, ineq, val
            labWidth <- max(sapply(names(list_ref$df), nchar))
            lab <- ttklabel(parent, text=nm, width=labWidth)
            ##
            inLab <- ttklabel(parent, text="%in%")
            ##
            btn <<- ttkbutton(parent, text=ellipsize(),
                              command=.self$select_values_dialog)
            ##
            sapply(list(lab, inLab), tkpack,
                   side="left", padx=5)
            tkpack(btn, expand=TRUE, fill="x", side="left")
            callSuper(parent)
          })
```

We leave it as an exercise for the reader to add a subclass for logical variables or date variables.

Editable tables of data

There is no native widget for editing the cells of tabular data, as is provided by the `edit` method for data frames. The `tktable` widget (`http://tktable.sourceforge.net/`) provides such an add-on to the base Tk. We don't illustrate its usage here, as we keep to the core set of functions provided by Tk. An interface for this Tcl package is provided in the `tcltk2` package (`tk2edit`). The `gdf` function of `gWidgetstcltk` is based on this.

Hierarchical data

Specifying tree-like or hierarchical data is nearly identical to specifying rectangular data for the `ttktreeview` widget. The widget provides column `#0` to display this extra structure. If an item, except the root, has children, a trigger icon to expand the tree is shown. This is in addition to any text and/or an icon that is specified. Children are displayed in an indented 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 3.9: 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"`.

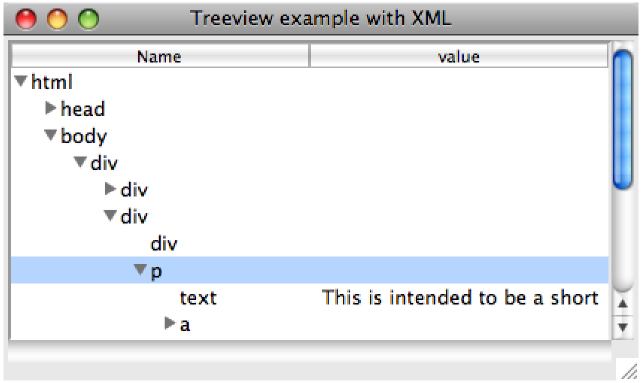

Figure 3.7: Illustration of using `ttktreeview` widget to show hierarchical data returned from parsing an HTML document with the `XML` package.

```
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)
}
```

3.5 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 feature was at one time common in GUIs, but now is rarely seen so it is recommended that this option be disabled. The `tearoff` option can be used at construction time to override the default behavior. Otherwise, the following command will do so globally:

```
tcl("option","add","*tearOff", 0)     # disable tearoff menus
```

A 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. Action commands may possibly be called for different widgets, so the use of percent substitution is problematic. 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 3.10: 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 begin with a simple GUI comprised of a top-level window containing the text entry widget.

```
w <- tktoplevel()
tkwm.title(w, "Simple code editor")
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
tb <- tktext(f, undo=TRUE)
addScrollbars(f, tb)
```

Using `tkmenu`, we create a toplevel menu bar, `mb`, and attach it to our toplevel window. Following we makefile and edit submenus.

```
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 addd an accelerator entry to the menubar and a binding to the top-level window for the keyboard shortcut for "undo."

```
if(usingMac()) {
  tkentryconfigure(editMenu, "Undo", accelerator="Cmd-z")
  tkbind(w,"<Option-z>", function() tcl(tb,"edit","undo"))
```

```
} else {
  tkentryconfigure(editMenu, "Undo", accelerator="Control-u")
  tkbind(w,"<Control-u>", function() tcl(tb,"edit","undo"))
}
```

To illustrate popup menus, we define one within our text widget that will grab all functions that complete the current word, using the `CompletePlus` function from the `svMisc` package to provide the completions. The use of `current wordstart` and `current wordend`, below, to find the word at the insertion point isn't quite right for R, as it stops at periods, but we don't pursue fixing this.

```
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
}
```

3.6 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 3.11: Using a canvas to make a scrollable frame
This example[8] 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))
```

[8] This example is modified from an example found at `http://mail.python.org/pipermail/python-list/1999-June/005180.html`

```
    if(gpwidth < width)
      tkitemconfigure(canvasWidget, gpID, width=width)
  })
  return(gp)
}
```

To use this, we create a simple GUI as follows:

```
w <- tktoplevel()
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 1.4). 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 3.12: 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.

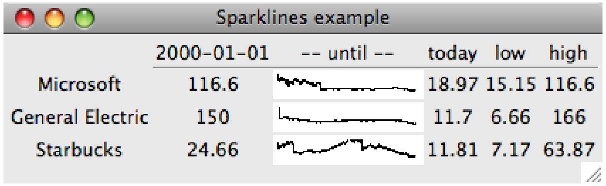

Figure 3.8: Example of embedding sparklines in a display organized using tkgrid. A tkcanvas widget is used to display the graph.

```
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(seq_along(x), function(i) {
    l[[2*i + 2]] <<- x[i]
    l[[2*i + 3]] <<- y1[i]
  })
  do.call("tcl",l)
```

```
  tkgrid(mL(label),mL(start), 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 3.13: 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"))
```