M. A. Newton, C. Czado, and R. Chappell. Bayesian inference for semiparametric binary regression. *Journal of the American Statistical Association* 91:142–153, 1996.

M. Plummer, N. Best, K. Cowles, and K. Vines. CODA: Output analysis and diagnostics for MCMC. *R package version 0.12-1*, 2007.

B. J. Smith. boa: Bayesian Output Analysis Program (BOA) for MCMC. *R package version 1.1.6-1*, 2007.

S. D. Spiegelhalter, N. G.Best, B. P. Carlin, and A. Van der Linde. Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society: Series B* 64:583–639, 2002.

*Alejandro Jara*
*Biostatistical Centre*
*Catholic University of Leuven*
*Leuven, Belgium*
`Alejandro.JaraVallejos@med.kuleuven.be`

# An Introduction to gWidgets

*by John Verzani*

## Introduction

CRAN has several different packages that interface R with toolkits for making graphical user interfaces (GUIs). For example, among others, there are **RGtk2** (Lawrence and Temple Lang, 2006), **rJava**, and **tcltk** (Dalgaard, 2001). These primarily provide a mapping between library calls for the toolkits and similarly named R functions. To use them effectively to create a GUI, one needs to learn quite a bit about the underlying toolkit. Not only does this add complication for many R users, it can also be tedious, as there are often several steps required to set up a basic widget. The **gWidgets** package adds another layer between the R user and these packages providing an abstract, simplified interface that tries to be as familiar to the R user as possible. By abstracting the toolkit it is possible to use the **gWidgets** interface with many different toolkits. Although, open to the criticism that such an approach can only provide a least-common-denominator user experience, we'll see that **gWidgets**, despite not being as feature-rich as any underlying toolkit, can be used to produce fairly complicated GUIs without having as steep a learning curve as the toolkits themselves.

As of this writing there are implementations for three toolkits, **RGtk2**, **tcltk**, and **rJava** (with progress on a port to **RwxWidgets**). The **gWidgetsRGtk2** package was the first and is the most complete. Whereas **gWidgetstcltk** package is not as complete, due to limitations of the base libraries, but it has many useful widgets implemented. Installation of these packages requires the base toolkit libraries be installed. For **gWidgetstcltk** these are bundled with the windows distribution, for others they may require a separate download.

## Dialogs

We begin by loading the package. Both the package and at least one toolkit implementation must be installed prior to this. If more than one toolkit implementation has been installed, you will be queried as to which one to use.

```
library("gWidgets")
```

The easiest GUI elements to create are the basic dialogs (Figure 1). These are useful for sending out quick messages, such as: [1]

```
gconfirm("Are we having fun?")
```

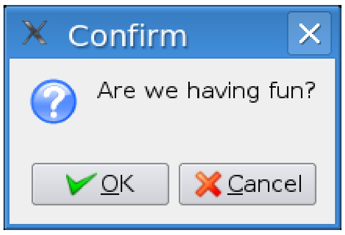


Figure 1: Simple dialog created by `gconfirm` using the **RGtk2** toolkit.

A basic dialog could be used to show error messages

```
options(error = function() {
  err = geterrmessage()
  gmessage(err, icon="error")
})
```

or, be an alternative to `file.choose`

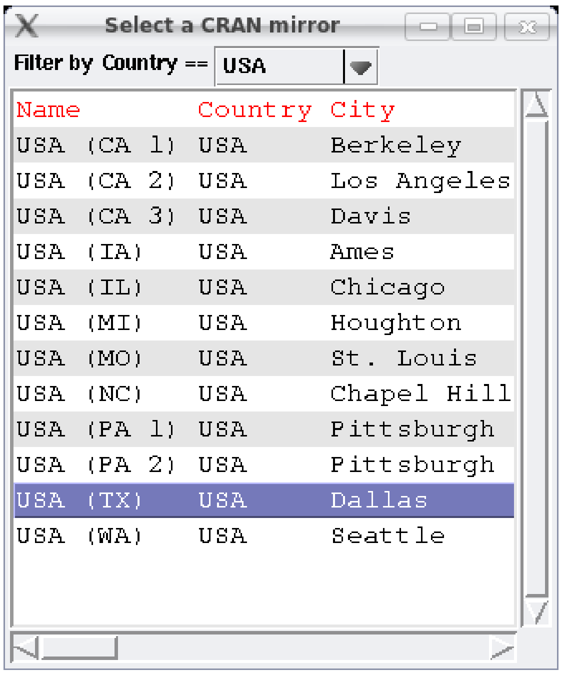
```
source(gfile())
```

In **gWidgets**, these basic dialogs are modal, meaning the user must take some action before control of R is returned. The return value is a logical or string, as appropriate, and can be used as input to a further command. Modal dialogs can be confusing

[1]The code for these examples is available from `http://www.math.csi.cuny.edu/pmg/gWidgets/rnews.R`

if the dialog gets hidden by another window. Additionally, modal dialogs disrupt a user's flow. A more typical GUI will allow the R session to continue and will only act when a user initiates some action with the GUI by mouse click, text entry, etc. The GUI designer adds handlers to respond to these events. The **gWidgets** programming interface is based around facilitating the following basic tasks in building a GUI: constructing widgets that a user can control to affect the state of the GUI, using generic functions to programmatically manipulate these widgets, simplifying the layout of widgets within containers, and facilitating the assigning of handlers to events initiated by the user of the GUI.

## Selecting a CRAN site



Filter by Country == USA

| Name | Country | City |
|---|---|---|
| USA (CA 1) | USA | Berkeley |
| USA (CA 2) | USA | Los Angeles |
| USA (CA 3) | USA | Davis |
| USA (IA) | USA | Ames |
| USA (IL) | USA | Chicago |
| USA (MI) | USA | Houghton |
| USA (MO) | USA | St. Louis |
| USA (NC) | USA | Chapel Hill |
| USA (PA 1) | USA | Pittsburgh |
| USA (PA 2) | USA | Pittsburgh |
| USA (TX) | USA | Dallas |
| USA (WA) | USA | Seattle |

Figure 2: GUI to select a CRAN mirror shown using **gWidgetstcltk**. The filter feature of `gtable` has been used to narrow the selection to USA sites. Double clicking a row causes the CRAN repository to be set.

Selecting an item from a list of items or a table of items is a very common task in GUIs. Our next example presents a GUI that allows a user to select with the mouse a CRAN repository from a table. The idea comes from a **tcltk** GUI created by the `chooseCRANmirror` function. This example shows how the widget constructors allow specification of both containers and event handlers.

We will use this function to set the CRAN repository from a URL.

```
setCRAN <- function(URL) {
  repos = getOption("repos")
  repos["CRAN"] <- gsub("/$", "", URL)
  options(repos=repos)
}
```

To begin our GUI we first create a top-level window to contain our table widget.

```
win <- gwindow("Select a CRAN mirror")
```

A call to `gwindow` will pop-up a new window on the screen and return an object which can be manipulated later. The argument sets the window title.

The widget we create will show the possible CRAN values in a tabular format for easy selection by a mouse. Selection occurs when a row is clicked. We will assign this function to respond to a double click event:

```
handler = function(h,...) {
  URL <- svalue(tbl) # get value of widget
  setCRAN(URL)       # set URL
  dispose(win)       # close window
}
```

The `svalue` function is a new generic function which retrieves the selected value for a widget, in this case one called `tbl` constructed below. The `dispose` function is a new generic which for windows causes them to be destroyed. So this handler will set the chosen URL and then destroy the top-level window.

The `gtable` function constructs a widget allowing the user to select a value from a data frame and returns an object for which generic functions are defined. The row of the value is selected with a mouse click, whereas, the selected column is specified at the time the widget is constructed using the `chosencol` argument. Additionally, this function allows the specification of a column whose unique values can be used to filter the display of possible values to select from.

```
tbl <- gtable(
  items=utils::getCRANmirrors(),
  chosencol=4,
  filter.column=2,
  container=win,
  handler=handler
)
```

Figure 2 shows the final widget using the **tcltk** toolkit.

## Using gWidgets

We now go through the basic steps of building a GUI using **gWidgets** a little more systematically.

### Containers

As illustrated previously a top-level window is created using `gwindow`. The **gWidgets** package only allows one widget to be packed into a top-level window. As such, typically a container that can hold more than one object is packed into the top-level window. There are several types of containers available.

The ggroup function produces a container that may be visualized as a box that allows new widgets to be packed in from left to right (the default) or from top to bottom (horizontal=FALSE). Nesting such containers gives a wide range of flexibility.

Widgets are added to containers through the container argument at the time of construction (which hereafter we shorten to cont) or using the add method for containers. However, **gWidgetstcltk** requires one to use the container argument when a widget is constructed. That is, except with **gWidgetstcltk**

```
win <- gwindow("test")
b <- gbutton("click me", cont=win)
```

is equivalent to

```
win <- gwindow("test")
add(win, gbutton("click me"))
```

The latter is a bit more convenient as it allows one to keep separate the widget's construction and its layout, but the former allows more portability of code.

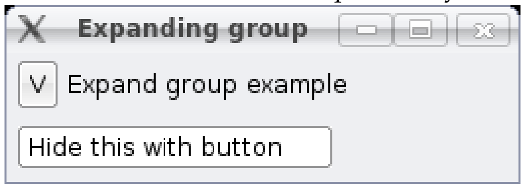


Figure 3: Expanding group example after resizing window. The button, label and text widgets use nested ggroup containers for their layout.

To illustrate nested ggroup containers and the add and delete generics for containers, the following example shows how to create a widget that can hide or show its contents when a button is clicked. This is more fully implemented in the gexpandgroup container.

We begin by defining a top-level window and immediately add a ggroup instance which packs in its widgets from top to bottom.

```
win <- gwindow("Expanding group")
g <- ggroup(horizontal=FALSE, cont=win)
```

Inside of g we add a nested group which will contain a button and a label.

```
g1 <- ggroup(horizontal=TRUE, cont=g)
button <- gbutton("V",cont=g1)
label <- glabel("Expand group example",
  cont=g1)
```

Finally, we add a ggroup instance to the g container to hold the widgets that we wish to hide and put a widget into the new container.

```
g2 <- ggroup(cont=g, expand=TRUE)
e <- gedit("Hide this with button",
 cont=g2)
```

That finishes the layout. Figure 3 shows the GUI in the expanded state. To make this example do something interesting, we define functions to respond to clicking on the button or the label. These functions toggle whether or not the g2 container is packed into the g container using the add and delete generics for containers.

```
expandGroup = function()
  add(g,g2, expand=TRUE)
hideGroup = function() delete(g,g2)
```

The add generic is used to add a widget (or container) to a container. In this case, the argument expand=TRUE is given to force the added widget to take up as much space as possible. The delete generic removes a widget from a container. The widget is not destroyed, as with dispose, just removed from its container.

Next, we define a handler to be called when the button or label is clicked. This involves arranging a means to toggle between states and to adjust the button text accordingly.

```
state <- TRUE  # a global
changeState <- function(h,...) {
  if(state) {
    hideGroup()
    svalue(button) <- ">"
  }  else {
    expandGroup()
    svalue(button) <- "V"
  }
  state <<- !state
}
```

We used the <<- operator so that the value of state is updated in the global environment, rather than the environment of the handler.

Finally, we bind this handler to both the button and the label, so that it is called when they are clicked.

```
ID <- addHandlerClicked(button,
  handler=changeState)
ID <- addHandlerClicked(label,
  handler=changeState)
```

There are just a few options for the layout of widgets within a ggroup container, compared to those available in the various toolkits. We illustrated the expand argument. With rJava and tcltk there is also an anchor argument which takes a value of the form c(x,y) where x or y are −1, 0, or 1. These are used to specify which corner of the space allocated by the container the widget should add itself in.

Additionally, the ggroup instances have methods addSpace to add a fixed amount of blank space and addSpring to add a "spring" which forces the remaining widgets to be pushed to the right (or the bottom) of the containers. This simple example shows how these can be used to put buttons on the right side of a dialog. First we pack a ggroup instance into a top-level container.

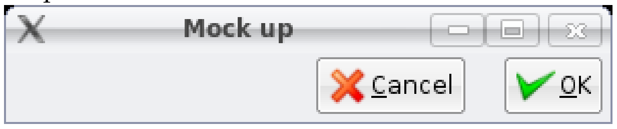


Figure 4: Illustration of `addSpace` and `addSpring` methods of `ggroup` using the **gWidgetsRGTk2** package after resizing main window. The buttons are pushed to the right side by `addSpring`.

```
win <- gwindow("Mock up")
bg <- ggroup(cont=win, expand=TRUE)
addSpring(bg)     # push to right
```

Now we add two buttons and a bit of space, 10 pixels, between them. By default, packing is done from left to right, starting on the left. The use of `addSpring` pushes the packed in containers and widgets to the right side of the container.

```
b1 <- gbutton("cancel", cont=bg)
addSpace(bg,10)   # add some space
b2 <- gbutton("ok", cont=bg)
```

The `expand=TRUE` value for `bg` ensures that that container expands to fill the space it can, otherwise the button widgets would push to the boundary of the button group and not the outside container `win`. Figure 4 shows the widget.

Other containers are illustrated in the examples below. These include `glayout` for laying out widgets in a grid, `gnotebook` for holding multiple widgets at once with tabs used to select between the widgets, and `gpanedgroup` which uses a movable pane to allocate space between widgets. Additionally, `gframe` is similar to `ggroup`, but with an external frame and area for a label.

## Basic widgets

A GUI is composed of widgets packed into containers. In **gWidgets** there are a number of familiar widgets. We've illustrated labels and buttons (`glabel` and `gbutton`). Additionally there are widgets to select from a list of items (`gradio`, `gdroplist`, and `gcheckboxgroup`); widgets to select from a table of values (`gtable`); widgets to select files or dates (`gfile`, `gcalendar`); widgets to edit text (`gedit`, `gtext`); a widget to edit a data frame (`gdf`); a widget to display graphic files (`gimage`); and others. Not all are available for each toolkit.

The following example, modified from a recent R News article on the **rpanel** package (Bowman et al., 2006), illustrates how many of these widgets can be combined to create a GUI to demonstrate confidence intervals. This example uses the `glayout` container to lay out its widgets in a grid.

First we pack the container into a top level window.

```
win <- gwindow("CI example")
tbl <- glayout(cont=win)
```

The `glayout` container uses matrix-like notation to specify where the widgets are placed. When the assigned value is a string, a `glabel` is used.

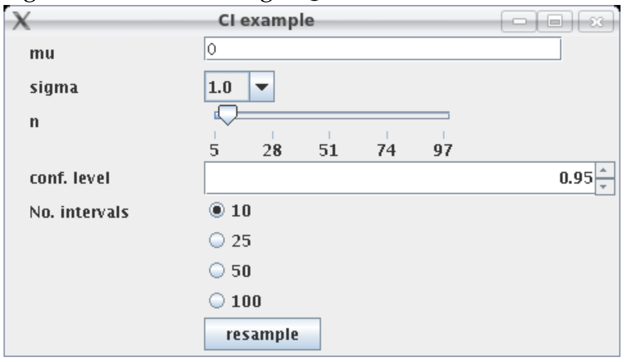


Figure 5: GUI for a confidence interval demonstration, using **gWidgetsrJava**, illustrating several different widgets: `gedit` for single-line editing; and `gdroplist`, `gslider`, `gspinbox`, and `gradio` for selection from a fixed set of items.

```
tbl[1,1] <- "mu"
tbl[1,2] <-
  (mu <- gedit("0",  cont=tbl,
    coerce.with=as.numeric))
tbl[2,1] <- "sigma"
tbl[2,2] <-
  (sigma <- gdroplist(c(1,5,10),
    cont=tbl))
tbl[3,1] <- "n"
tbl[3,2] <- (
  n <- gslider(from=5,to=100, by=1,
    value = 10, cont=tbl))
tbl[4,1] <- "conf. level"
tbl[4,2, expand=TRUE] <-
  (confLevel <-  gspinbutton(
    from=0.5, to=0.99, by=0.01,
    value=0.95, cont=tbl))
tbl[5,1] <- "No. intervals"
tbl[5,2] <-
  (noIntervals <- gradio(c(10,25,
    50,100), cont=tbl))
tbl[6,2] <-
  (resample <- gbutton("resample",
    cont=tbl))
visible(tbl) <- TRUE
```

The last line with `visible` is necessary for **gWidgetsRGtk2** to produce the layout.

The matrix-notation refers to where the widget is placed. An assignment like

```
tbl[1:2,2:3] <- "testing"
```

would place a label within the area for the first and second rows and second and third columns. Unlike matrices, the notation is not used to vectorize the placement of several widgets, extract widgets, or replace a widget. The above code completes the widget construction and layout. Figure 5 shows a realization using **rJava**.

The widgets have various arguments to specify their values that depend on what the widget does. For example, the slider and spinbutton select a value from a sequence, so the arguments mirror those of `seq`. Some widgets have a `coerce.with` argument which allows a text-based widget, like `gedit`, to return numeric values through `svalue`. The `gdroplist` widget is used above to pop up its possible values for selection. Additionally, if the `editable=TRUE` argument is given, it becomes a combo-box allowing a user to input a different value.

In Figure 5 we see in the slider widget one of the limitations of the **gWidgets** approach – it is not as feature rich as any individual toolkit. Although the underlying JSlider widget has some means to adjust the labels on the slider, **gWidgets** provides none in its API, which in this case is slightly annoying, as Java's chosen values of 5, 28, 51, 74, and 97 just don't seem right.

The values specified by these widgets are to be fed into the function `makeCIs`, which is not shown here, but uses `matplot` to construct a graphic displaying simulated confidence intervals. To make this GUI interactive, the following handler will be used to respond to changes in the GUI. We need to call `svalue` on each of the widgets and then pass these results to the `makeCIs` function. This task is streamlined by using `lapply` and `do.call`:

```
allWidgets <- list(mu,sigma,noIntervals,
  n, confLevel, resample)
plotCI <- function(h, ...) {
  lst <- lapply(allWidgets,svalue)
  do.call(makeCIs,lst)
}
```

Again we use `lapply` to add the handler to respond to the different widget default events all at once.

```
invisible(sapply(allWidgets,function(i)
  addHandlerChanged(i,handler=plotCI))
)
```

## Interacting with widgets programmatically

In **gWidgets**, the widget and container constructors return S4 objects of various classes. These objects are accessed programmatically with generic functions. As was sensible, generic methods familiar to R users were used, and when not possible, new generics were added. For example, for widgets which store values to select from, the generics `[` and `[<-` are used to refer to the possible values. For other widgets, as appropriate, the generics `dim`, `names`, `length`, etc., are defined. To get and set a widget's "selected" value, the new generic functions `svalue` and `svalue<-` were defined. The term "selected" is loosely defined, e.g., applying to a button's text when clicked.

This simple example shows how selecting values in a checkbox group can be used to set the possible values in a drop list. First we create a container to hold the two widgets.

```
win <- gwindow("methods example")
g <- ggroup(cont=win)
```

The construction of the two widgets is straightforward: simply specify the available values using `items`.

```
cb <- gcheckboxgroup(items=letters[1:5],
  cont=g)
dl <- gdroplist(items="", cont=g)
```

To finish, the following adds a handler to `cb` which will update the possible values of `dl` when the user changes the value of the checkbox group.

```
ID <- addHandlerChanged(cb,
 function(h,...) {
   dl[] <- svalue(cb)
})
```

A few other new generics are defined in **gWidgets** for the basic widgets such as `font<-` for setting font properties, `size<-` to adjust the size in pixels for a widget, `enabled<-` to adjust if the widget can accept input, and the pair `tag` and `tag<-`, which is like `attr` only its values are stored with the widget. This latter function is useful as its values can be updated within a function call, such as a handler, as this next example shows. The expanding group example with the `state` variable would have been a good place to use this feature.

```
> x = gbutton("test", cont=gwindow())
> tag(x,"ex") <- attr(x,"ex") <- "a"
> f = function(y)
+   tag(y,"ex") <- attr(y,"ex") <- "b"
> c(tag(x,"ex"),attr(x,"ex"))

[1] "a" "a"

> f(x)
> c(tag(x,"ex"),attr(x,"ex"))

[1] "b" "a"
```

## Handlers

A GUI user initiates an event to happen when a button is clicked, or a key is pressed, or a drag and drop is initiated, etc. The **gWidgets** package allows one to specify a handler, or callback, to be called when an event occurs. These handlers may be specified when a widget is constructed using the `handler` argument, or added at a later time. To add the default handler, use the `addHandlerChanged` generic. Most, but not all, widgets have just one type of event for which a handler may be given. For instance, the `gedit` widget has the default handler respond to the event of a user pressing the ENTER key. Whereas, the `addHandlerKeystroke` method can be used to add a handler that responds to any keystroke. Officially, only one handler can be assigned per event, although some toolkits allow more.

Handlers return an ID which can be used with `removeHandler` to remove the response to the event.

The signature of a handler function has a first argument, `h`, to which is passed a list containing components `obj`, `action` and perhaps others. The `obj` component refers to the underlying widget. In the examples above we found this widget after storing it to a variable name, this provides an alternative. The `action` component may be used to passed along an arbitrary value to the handler. The other components depend on the widget. For the `ggraphics` widget (currently just **gWidgetsRGtk2**), the components `x` and `y` report the coordinates of a mouse click for `addHandlerClicked`. Whereas, for `addDropTarget` the `dropdata` component contains a string specifying the value being dragged.

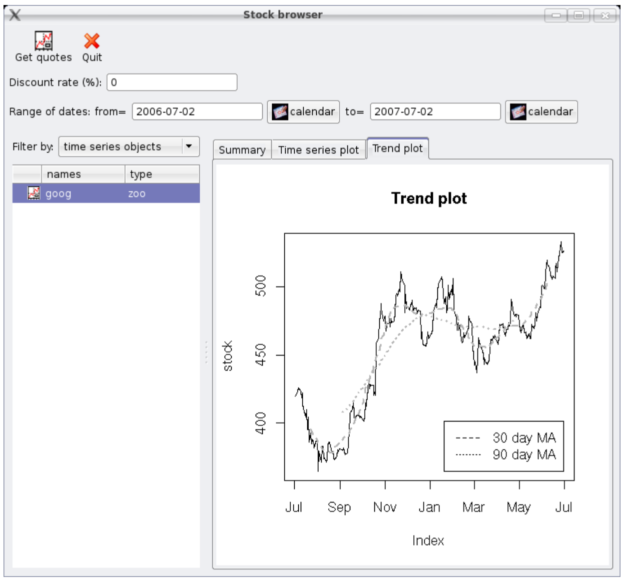
To illustrate the drag-and-drop handlers the following creates two buttons. The drop source needs a handler which returns the value passed along to the `dropdata` component of the drop target handler. In this example, clicking on the first button and dragging its value onto the label will change the text displayed on the label.

```
g = ggroup(cont=gwindow("DnD example"))
l1 <- gbutton("drag me", cont=g)
l2 <- glabel("drop here", cont=g)
ID <- addDropSource(l1, handler=
   function(h,...) svalue(h$obj))
ID <- addDropTarget(l2, handler =
  function(h,...)
   svalue(h$obj) <- h$dropdata)
```

Drag and drop works with all the toolkits except **gWidgetsrJava** where only what is provided natively through the Java libraries is working.

## An application

This last example shows how to make a more ambitious GUI for a task; in this case, an interface to download and visualize stock prices. New widgets and containers shown are the toolbar widget; the notebook container, which is used to present different simultaneous summaries of a stock; the variable selector widget, for selecting variables from the workspace; and the generic widget constructor, which creates a simple GUI based on a function's arguments, as found by `formal`. Although more involved than the examples above, it is meant to show how specialized GUIs can be formed relatively easily by gluing together the pieces available through **gWidgets**. Figure 6 shows the result of running the following code, then downloading data for Google into the session.



Discount rate (%): 0
Range of dates: from= 2006-07-02 to= 2007-07-02
Filter by: time series objects

| names | type |
|---|---|
| goog | zoo |

Figure 6: Stock browser widget using **gWidgetsRGtk2**. This illustrates a toolbar, the variable selector, and a notebook container.

We begin by loading the **tseries** package which contains the `get.hist.quote()` function for retrieving stock data from `www.yahoo.com` and calls in the handy **zoo** package by Achim Zeileis and Gabor Grothendieck or use for later use.

```
invisible(require("tseries"))
```

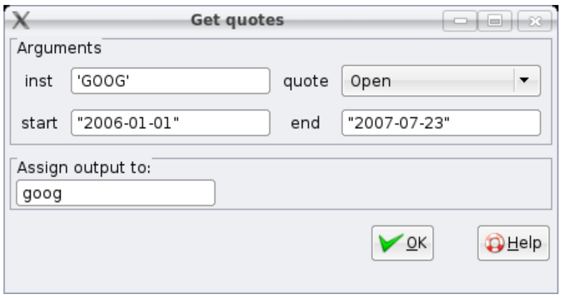
In previous examples we used the global environment to store our variables. This works well for a single GUI, but can quickly lead to problems with colliding variable names with multiple GUIs. Setting up a separate environment for each GUI is one way around this potential conflict. Similarly, the **proto** package can be used effectively to provide an environment and more. Other possible methods are to use namespaces or the environment within a function body. In this example, we create a new environment to store our global variables.

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

The environment `e` is accessed using list-like syntax.

To download stock information we use `get.hist.quote`. We define this function to reduce the number of arguments so that `ggenericwidget` produces a simpler GUI (Figure 7).

```
getQuotes = function(
  inst = "GOOG",
  quote = c("Open","High","Low","Close"),
  start, end)
get.hist.quote(inst, start=start,
  end=end, quote=quote)
```



inst 'GOOG' quote Open
start "2006-01-01" end "2007-07-23"
Assign output to: goog

Figure 7: The GUI for getQuotes created by ggenericwidget using the **RGtk2** toolkit.

We define two functions to display graphical summaries of the stock, using the **zoo** package to produce the plots. Rather than use a plot device (as only **gWidgetsRGtk2** implements an embeddable one through ggraphics), we choose instead to make a file containing the plot as a graphic and display this using the gimage widget. In **gWidgetstcltk** the number of graphic formats that can be displayed is limited, so we use an external program, convert, to create a gif file. This is not needed if using **RGtk2** or **rJava**.

```
showTrend <- function(stock) {
  trendPlotFile <- e$trendPlotFile
  png(trendPlotFile)
  plot(stock, main="Trend plot")
  lines(rollmean(stock, k = 30), lwd=2,
    lty=2, col=gray(.7))
  lines(rollmean(stock, k = 90), lwd=2,
    lty=3, col=gray(.7))
  legend(index(stock)[length(stock)],
   min(stock),
   legend=c("30 day MA","90 day MA"),
   lty=2:3,xjust=1, yjust=0)
  dev.off()
  ## create a gif file
  system(paste("convert ",
    trendPlotFile," ",
    trendPlotFile,".gif",sep=""))
  svalue(e$trendPlot) = paste(
    trendPlotFile,".gif",sep="")
}
```

The function showDiscount is similarly defined but not shown.

The main function below is used to update the notebook pages when a new stock is selected, or the graphing parameters are changed. This function refers to widgets defined in a subsequent code chunk.

```
updateNB <- function(h,...) {
  ## update data set
  ifelse(e$curDSName != "",
         e$curDS <- get(e$curDSName),
         return())
  ## get data within specified range
  start <- svalue(e$startD)
  if(start == "") start=NULL
  end <- svalue(e$endD)
  if(end == "") end = NULL
  dataSet = window(e$curDS,
    start=start,end=end)
  ## update summaries
  svalue(e$noSum) =
    capture.output(summary(dataSet))
  showDiscount(dataSet,
    svalue(e$discRate))
  showTrend(dataSet)
}
```

Now the main application is defined. First some variables.

```
e$curDSName <- ""
e$curDS <- NA
e$tsPlotFile <- tempfile()
e$trendPlotFile <- tempfile()
```

Next we define the main window and a container to hold the subsequent widgets.

```
## layout
e$win <- gwindow("Stock browser")
e$gp <- ggroup(horizontal=FALSE,
  cont=e$win)
```

The first widget will be a toolbar. This basic one has two actions, one to open a dialog allowing a user to download stock information from the internet, the other a quit button. To be more friendly to R users, The gtoolbar() constructor uses a list to specify the toolbar structure, rather than, say, XML. Basically, each named component of this list should be a list with a handler component and optionally an icon component. Menubars are similarly defined, only nesting is allowed.

```
## Set up simple toolbar
tb <- list()
tb$"Get quotes"$handler <- function(...)
  ggenericwidget("getQuotes",
    container=gwindow("Get quotes"))
tb$"Get quotes"$icon <- "ts"
tb$Quit$handler <- function(...)
  dispose(e$win)
tb$Quit$icon <- "cancel"
theTB <- gtoolbar(tb, cont=e$gp)
```

This application has widgets to adjust the interest rate (used as a discounting factor), and the start and end dates. The date widgets use the gcalendar() constructor for easier date selection.

```
## Now add parameters
e$pg <- ggroup(horizontal=TRUE,
  cont = e$gp)
## the widgets
l <- glabel("Discount rate (%):",
  cont=e$pg)
e$discRate <- gedit(0, cont=e$pg,
    coerce.with=as.numeric,
    handler=updateNB)
  e$pg <- ggroup(horizontal=TRUE,
    cont = e$gp)
  l <- glabel("Range of dates:",
    cont=e$pg)
  curDate <- Sys.Date()
  l <- glabel("from=", cont=e$pg)
  e$startD <-
    gcalendar(as.character(curDate-365),
    handler=updateNB, cont=e$pg)
  l <- glabel("to=", cont=e$pg)
  e$endD <-
    gcalendar(as.character(curDate),
    handler=updateNB, cont=e$pg)
```

A paned group is used for the final layout, alllowing the user to adjust the amount of room for each major part of the GUI. One part is a variable selection widget provided by `gvarbrowser()`. Double clicking a row calls the `updateNB` handler to update the notebook pages.

```
  e$gpg <- gpanedgroup(cont=e$gp,
    expand=TRUE)
  e$varSel <- gvarbrowser(
    cont= e$gpg,
    handler = function(h,...) {
      e$curDSName <- svalue(e$varSel)
      updateNB()
    })
```

The other major part of the GUI is a notebook container to hold the three different summaries of the stock.

```
  e$nb <- gnotebook(cont=e$gpg)
```

Finally, we add pages to the notebook container below. The additional `label` argument sets the text for the tab. (These can be adjusted later using `names<-`.)

```
  ## A numeric summary of the data set
  e$noSum <- gtext("Numeric summary",
    cont=e$nb, label="Summary")
  ## First graphic summary
  e$tsPlot <- gimage(e$tsPlotFile,
    cont=e$nb, label="Time series plot")
  size(e$tsPlot)  <- c(480,480)
  ## secondGraphicSummary
  e$trendPlot <- gimage(e$trendPlotFile,
    cont=e$nb, label="Trend plot")
```

## Acknowledgments

The **gWidgets** package was designed as an extension of the **iWidgets** package (Urbanek, 2007) of Simon Urbanek for Michael Lawrence's **RGtk2** package (an extension of Duncan Temple Lang's **RGtk** package). Michael had done an initial port. Philippe Grosjean suggested to the author to abstract the code to make it independent of toolkit and Simon Urbanek contributed several key ideas for this. The author would like to thank all of the above for helpful comments, and the thoughtful and thorough report of an anonymous referee of an initial version of this article.

## Bibliography

A. Bowman, E. Crawford, and R. Bowman. rpanel: Making graphs move with **tcltk**. *R News*, 6(5):12–17, October 2006.

P. Dalgaard. The R-Tcl/Tk interface. In F. Leisch and K. Hornik, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing (DSC 2001)*, March 2001. ISSN: 1609-395X.

M. Lawrence and D. Temple Lang. RGTK2–A GUI Toolkit for R. *Statistical Computing and Graphics*, 17(1), 2006. Pusblished at `http://www.amstat-online.org/sections/graphics/newsletter/newsletter.html`.

S. Urbanek. iWidgets. `http://www.rforge.net/iWidgets/`.

*John Verzani*
*The College of Staten Island*
*City University of New York*
`verzani@math.csi.cuny.edu`