

gWidgets: a Toolkit-Independent API for Building GUIs in R

John Verzani

CUNY/The College of Staten Island

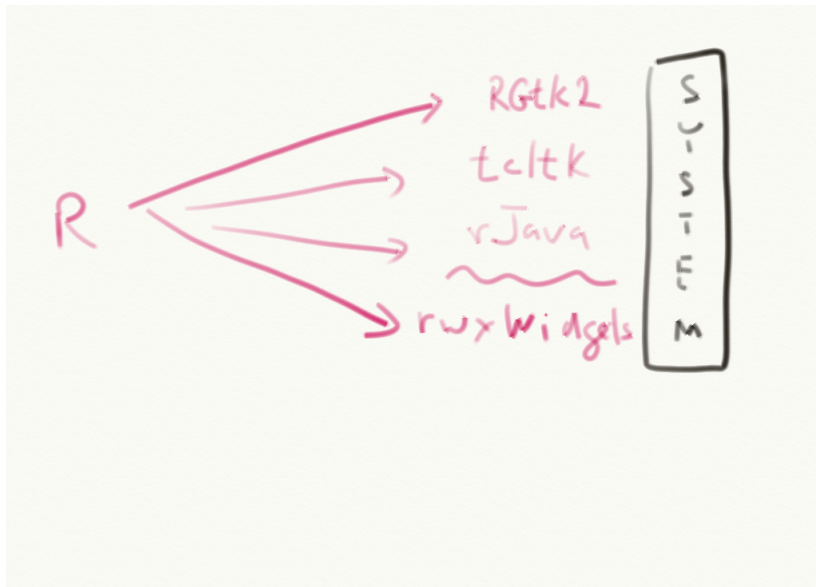
useR!2007

The role of gWidgets

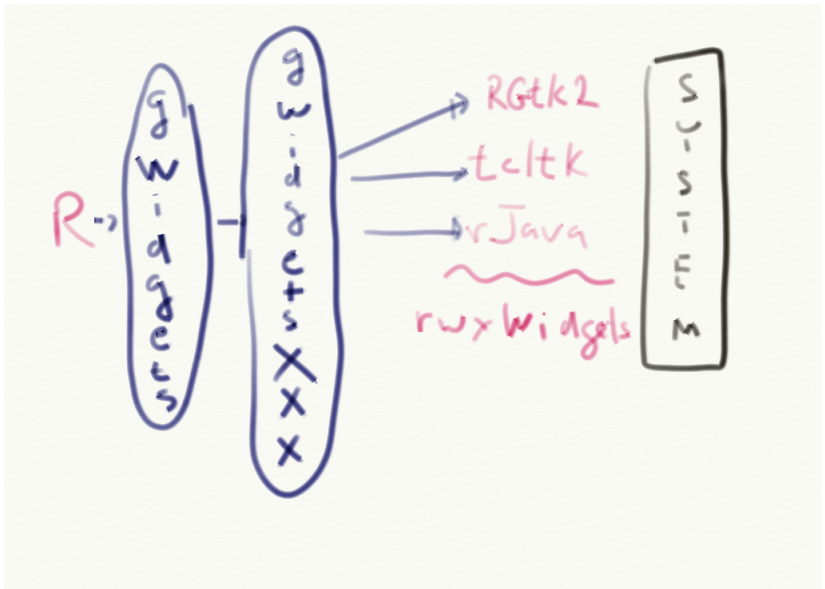
R has several packages (RGtk2, tcltk, rJava, RwxWidgets, ...) that allow the R user to interface with GUI toolkits.

The gWidgets package provides a *toolkit-independent* means to interface with these toolkits using an *simplified* programming interface.

The role of gWidgets (cont.)



The role of gWidgets (cont.)



Benefits and costs

Benefits

- The API is *simpler* than the toolkit APIs. “Rapid development”
- Some R specific features make building R GUIs easier
- portable to other toolkits

Costs

- The API is *simpler* TRANSLATION **not nearly as powerful:** lowest common denominator,...
- The cross-toolkit portability has some wrinkles – particularly with layout – that need ironing out.

Target users

Target users are R users who do not have detailed knowledge of a GUI toolkit, and do not want learn, but do want to make a relatively interactive GUI with out much fuss.

History

- The basic idea came from the `iWidgets` package of Simon Urbanek. This was designed for `rJava` and `JGR`. Simon also provided key insights into extending the package to other toolkits.
- Michael Lawrence ported `iwidgets` as an example for his `RGtk2` package. (This package extended Duncan Temple Lang's `RGtk` package.)
- After seeing an early extension, `iWidgetsRGtk2`, Philippe Grosjean suggested, to me, making the language toolkit independent.

Starting gWidgets

The `gWidgets` package requires one or more of the toolkit packages to be installed. If there is more than one, then the `guiToolkit` option can be set to specify the default, e.g.:

Starting gWidgets

```
> options(guiToolkit = "RGtk2")  
> library(gWidgets)
```

Installation of the toolkit packages is straightforward **if** the necessary system libraries are installed, as the packages sit on CRAN. For example,

Installing gWidgets for RGtk2

```
> install.packages("gWidgetsRGtk2", dep=TRUE)
```

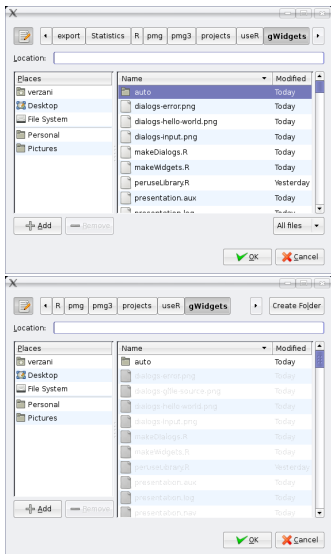
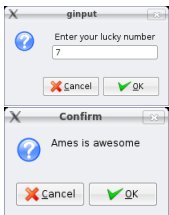
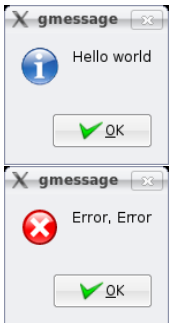
The basic dialogs

There are a few basic dialogs defined. In gWidgets the “basic dialogs” are modal (Require dismissal for other events to occur) and invisibly return either a logical or a string.

sample dialogs

```
gmessage("Hello world", title="gmessage")
gmessage("Error, Error", title="gmessage",
  icon="error")
ginput("Enter your lucky number",text="7",
  title="ginput", icon="question")
gconfirm("Ames is awesome", title="gconfirm")
source(gfile())
setwd(gfile(type="selectdir"))
```


sample dialog examples



The widgets

A GUI is usually non-modal. The gWidgets package is centered around 4 key aspects to creating a GUI:

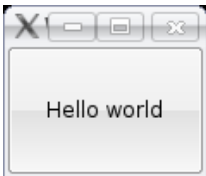
- Constructing *widgets* (easily)
- Programmatically interacting with the widgets in an R-like manner using *S4 methods*
- Facilitating adding *handlers* to respond to user-driven *events* in a GUI
- Facilitating the layout of widgets in *containers*

We next show most of the basic widgets to give a sense of the scope of the package. These are shown using all three toolkits if possible.

gbutton: clickable buttons

gbutton

```
gbutton("Hello world", cont=TRUE)
```



gbutton – some have icons

button with icon

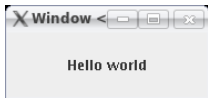
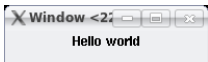
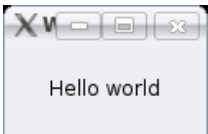
```
gbutton("ok", cont=TRUE)
```



glabel: adding clickable text

glabel

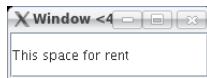
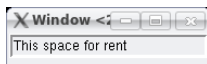
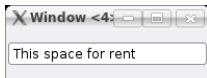
```
glabel("Hello world", cont=TRUE)
```



gedit: single line text

gedit

```
gedit("This space for rent", cont=TRUE)
```

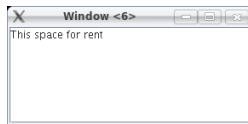
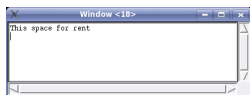
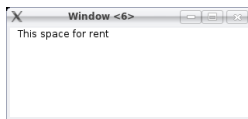


[In addition to the `gedit` widget for editable single-line text see also `glabel` with `editable=TRUE` and `and` `gdroplist`]

gtext: multi-line text

gtext

```
gtext("This space for rent", cont=TRUE)
```

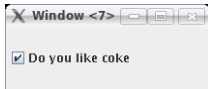
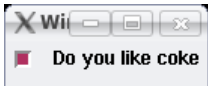
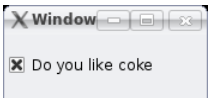


[Scrollbars are programmed in. Some toolkits only show them if needed.]

gcheckbox: Return TRUE or FALSE

gcheckbox

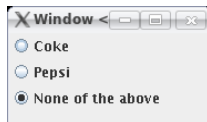
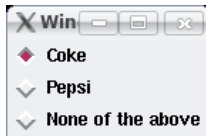
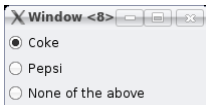
```
gcheckbox("Do you like coke", cont=TRUE)
```



gradio: Select one item

gradio

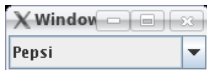
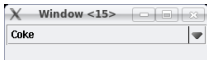
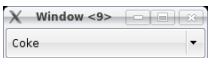
```
items = c("Coke", "Pepsi", "None of the above")  
gradio(items, cont=TRUE)
```



gdroplist : select one item

gdroplist

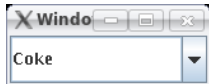
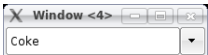
```
gdroplist(items, cont=TRUE)
```



gdroplist: select one item or enter your own (a combobox)

gdroplist: aka a combobox

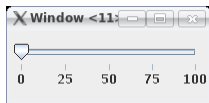
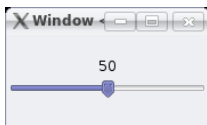
```
gdroplist(items, editable=TRUE, cont=TRUE)
```



gslider : select from a sequence

gslider

```
gslider(from=0, to = 100, by = 1, cont=TRUE)
```

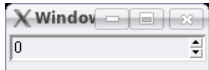
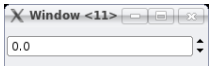


[In tcltk only integer selections are possible. No ability (currently) to adjust displayed axis values.]

gspinbutton: select from a sequence

gspinbutton

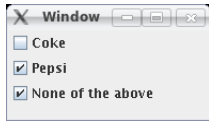
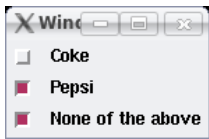
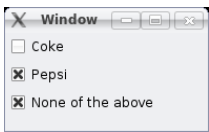
```
gspinbutton(from=0, to = 1, by = 0.1, cont=TRUE)
```



gcheckboxgroup: select none, one or more items

gcheckboxgroup

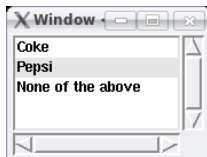
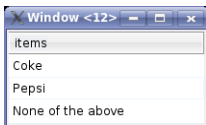
```
gcheckboxgroup(items, cont=TRUE)
```



`gtable`, with a vector argument. The `multiple=TRUE` argument allows multiple selections

`gtable`

```
gtable(items, multiple=TRUE, cont=TRUE)
```



[`tcltk` is "hacked together" as no underlying table widget in the base libraries; `rJava` has sizing and header issue.]

gtable, data frame argument. Allows selection by clicking on row.

gtable

```
gtable(mtcars,chosencol=6, cont=TRUE)
```

mpg	cyl	disp	hp	d
21.000000	6.000000	160.000000	110.000000	3
21.000000	6.000000	160.000000	110.000000	3
22.800000	4.000000	108.000000	93.000000	3
21.400000	6.000000	258.000000	110.000000	3
18.700000	8.000000	360.000000	175.000000	3

mpg	cyl	disp	hp	drat	wt
21.0	6	160.0	110	3.90	2.620
21.0	6	160.0	110	3.90	2.875
22.8	4	108.0	93	3.85	2.320
21.4	6	258.0	110	3.08	3.215
18.7	8	360.0	175	3.15	3.440

mpg	cyl	disp	hp	drat	wt
21	6	160	110	3.9	2.62
21	6	160	110	3.9	2.875
22.8	4	108	93	3.85	2.32
21.4	6	258	110	3.08	3.215
18.7	8	360	175	3.15	3.44
18.1	6	225	105	2.76	3.46
14.3	8	360	245	3.21	3.57
24.4	4	146.7	62	3.69	3.19
22.8	4	140.8	95	3.92	3.15

gdf: edit a data frame

```
gdf
require(MASS)
gdf(Cars93, cont=TRUE)
```

Row.names	Manufacturer	Model	Type	Min.Price	Pr
1	Acura	Integra	Small	12.900000	15
2	Acura	Legend	Midsize	29.200000	33
3	Audi	90	Compact	25.900000	29
4	Audi	100	Midsize	30.800000	37

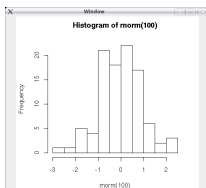
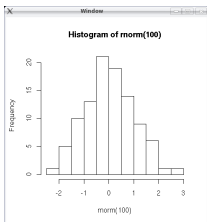
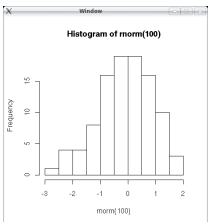
rownames	Manufactu...	Model	Type	Min.Price
1	Acura	Integra	Small	12.9
2	Acura	Legend	Midsize	29.2
3	Audi	90	Compact	25.9
4	Audi	100	Midsize	30.8
5	BMW	535i	Midsize	23.7
6	Buick	Century	Midsize	14.2

[Not available in tcltk (not in base set of libraries); double click to edit cell contents]

gimage: Show a graphic file

gimage

```
png("/tmp/rnorm.png")  
hist(rnorm(100))  
dev.off() ## tcltk has limited number of formats  
system("convert /tmp/rnorm.png /tmp/rnorm.gif")  
gimage("/tmp/rnorm.gif", cont=TRUE)
```



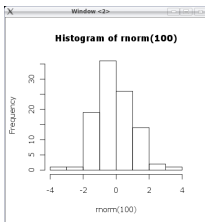
[Also can show some stock icons]

ggraphics: embeddable graphics device

ggraphics

```
> ggraphics(cont=TRUE)
> hist(rnorm(100))
> dev.list()
```

Cairo
2

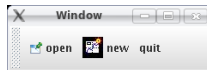
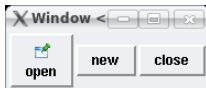


[RGtk2 only. rJava seems possible, tcltk has no embeddable device.]

gtoolbar: uses a list to define a toolbar

gtoolbar

```
f = function(h,...) print("Hello world")
tblst=list(
  open = list(handler=f, icon="open"),
  new  = list(handler=f, icon="new"),
  quit = list(handler=f, icon="close")
)
gtoolbar(tblst, cont=TRUE)
```

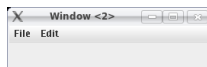
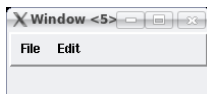
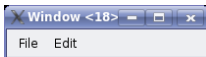


[RGtk2 looks best]

gmenu: maps a list to a menubar

gmenu

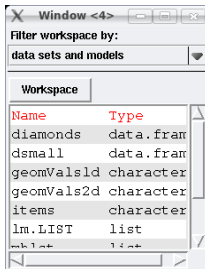
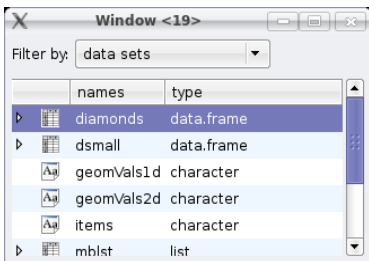
```
mblst = list(  
  File=list(  
    open = list(handler=f,icon="open"),  
    quit = list(handler=f,icon="close")  
  ),  
  Edit = list(  
    cut = list(handler=f),  
    copy = list(handler=f)  
  )  
)  
gmenu(mblst, cont=TRUE)
```



gvarbrowser: workspace browser

gvarbrowser

```
gvarbrowser(cont=TRUE)
```

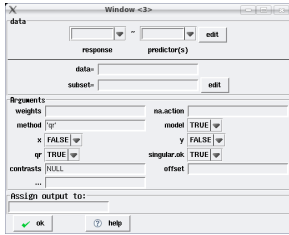
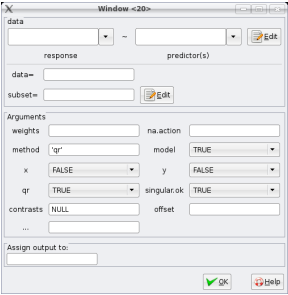


[In RGtk2 a gtree widget is used. This isn't implemented in the other toolkits]

ggenericwidget: create GUI from a function name using formals()

ggenericwidget

```
ggenericwidget("lm", cont=TRUE)
```



[Actually, ggenericwidget maps a list to a GUI. The list can be modified easily to tailor the GUI. These are non-interactive (must click OK button to proceed)]

Widget methods

A widget constructor returns S4 objects that can be manipulated using some standard generics and some new generics. For example for `gradio`

- the new generics `svalue` and `svalue<-` are used to get and set the *selected* value
- The familiar generics `[` and `[<-` are used to get or set the possible items to select from.

Widget methods (cont.)

```
b = gradio(c("coke","pepsi","neither"), cont=TRUE)
> svalue(b)                # retrieve value
[1] "coke"
> svalue(b) <- "pepsi"    # set by name
> svalue(b)
[1] "pepsi"
> svalue(b,index=TRUE) <- 1 # set by index
> svalue(b)
[1] "coke"
> b[1]                    # get label value
[1] "coke"
> b[1] <- "COKE"         # set label value
> svalue(b)
[1] "COKE"
```

Widget methods (cont.)

As appropriate, the following generics were used [, [<-, length, dim, names, names<- . Some new generics were added:

```
svalue      get selected value  
svalue<-   set selected value  
enabled<-  turn widget gray, disable input  
size<-     set widget size  
font<-     set widget font  
tag,tag<-  get, set attributes
```

[Also, some additional methods for containers are new.]

Event handlers

An interactive GUI has **handlers** which respond to **events** initiated by a user, eg. *a mouse click, pressing the enter key, pressing a key, a drag and drop* etc. In gWidgets each widget has a default handler. (eg. for buttons, a click; for gedit the enter key) A function can be specified at the time of construction using the handler argument. The action argument is passed to the handler.

```
widget_constructor(...stuff...,
  handler=function(h,...) {
    # h is a list h$obj -> widget, h$action -> action value
    ... function body ...
  },
  action = parameter,
  ...)
```

Handler examples

Handler examples

```
## print message
gbutton("press me", cont=TRUE, handler =
  function(h,...) print("hello world")
)
## change value
gbutton("press me hard", cont=TRUE, handler =
  function(h,...) svalue(h$obj) <- "Ouch, that hurt."
)
## gtable responds to double click
gtable(1:3, cont=TRUE, handler=
  function(h,...) print(svalue(h$obj))
)
```

the addHandlerXXX methods

Handlers can be added after a widget is constructed using the addHandlerXXX methods. The default handler is added using addHandlerChanged, where “changed” is loosely interpreted. Others are addHandlerClicked, addHandlerDoubleClick, addHandlerKeystroke, etc.. Most widgets have only one handler, but some have two or more. For instance, in a text widget there is a distinction between a keypress and the enter key.

Handler examples

```
e = gedit("text", cont=TRUE)
addHandlerChanged(e, function(h,...) print("changed"))
addHandlerKeystroke(e, function(h,...)
  print("keystroke"))
```

Perusing your Library of packages

We can now write a small application. This one uses the `gtable` widget to display all the installed packages in a user's library of packages. A handler responding to a double click will load or detach the package.

First a function to return a data frame of the installed packages

helper function

```
getPackages = function(...) {  
  allPackages = .packages(all.available=TRUE)  
  loaded = allPackages %in% .packages()  
  data.frame(Package=allPackages,loaded=loaded,  
    stringsAsFactors=FALSE)  
}
```

An application (cont.). Define handler

toggle package state handler

```
packageHandler = function(h,...) {  
  packages = svalue(h$obj, drop=FALSE) ## as d.f.  
  for(i in 1:nrow(packages)) {  
    if(packages[i,2] == TRUE) {  
      pkg = paste("package:",packages[i,1],sep="")  
      detach(pos = match(pkg, search()))  
    } else {  
      require(packages[i,1], character.only=TRUE)  
    }  
  }  
  h$obj[,] = getPackages() ## update list  
}
```

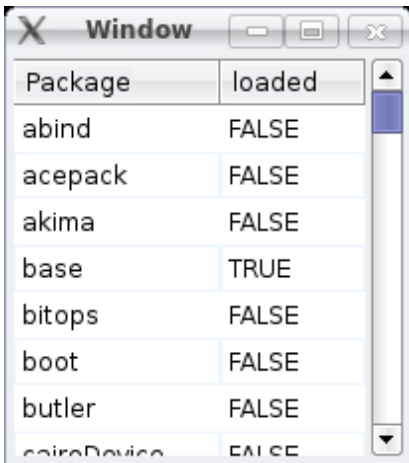
An application (cont.)

Finally, a call to `gtable`

create widget with event handler

```
packageList = gtable(  
  items = getPackages(),  
  sort.columns = 1:2,  
  handler = packageHandler,  
  container = TRUE  
)
```


An application (cont.)



The screenshot shows a window titled "Window" with standard window controls (minimize, maximize, close). Inside the window is a table with two columns: "Package" and "loaded". The table lists several packages and their loaded status. A vertical scrollbar is visible on the right side of the table.

Package	loaded
abind	FALSE
acepack	FALSE
akima	FALSE
base	TRUE
bitops	FALSE
boot	FALSE
butler	FALSE
cairoDevice	FALSE

Figure: Application to peruse library of packages. Load or detach package by double clicking of row.

An application: Filtering

If there are several packages, the `gtable` display can be filtered. Specifying a column number will allow filtering by the unique values of the column, but in this case we want to filter by the first letter.

Filter code

```
firstLetter = function(x)
  tolower(unlist(strsplit(x,""))[1])
filter.FUN = function(d, val ) {
  if(val == "")
    return(rep(TRUE, dim(d)[1]))
  else
    sapply(d[,1],firstLetter) == val
}
```

An application (cont.)

The `filter.FUN` and `filter.labels` arguments are used to add the filter

Add filter to widget

```
packageList = gtable(  
  items = getPackages(),  
  sort.columns = 1:2,  
  filter.labels = c("", letters),  
  filter.FUN = filter.FUN,  
  handler = packageHandler,  
  container = TRUE  
)
```

An application (cont.)

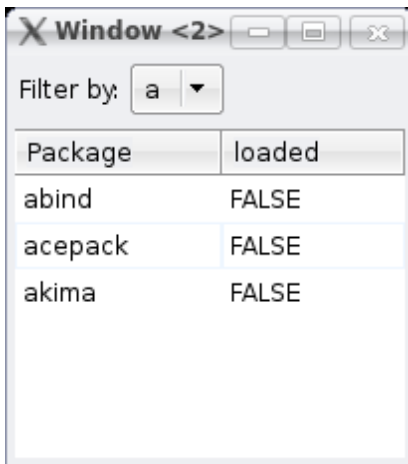


Figure: Application with addition of a filtering feature

Containers

More complicated applications require some idea of a container to pack widgets into. In `gWidgets` there is a distinction between a top-level window (produced with `gwindow`, or using `cont=TRUE`) and other containers.

The easiest to use container is the `ggroup` container. This container packs in its child widgets (and containers) from left to right (the default) or from top to bottom (when `horizontal=FALSE`.)

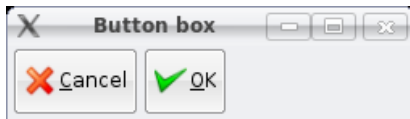
A widget or container is added using the `container` argument at the time of construction, or with the `add` method of containers.

A button box

Here is how to make a button box.

Button box (take 1)

```
win = gwindow("Button box")
g = ggroup(cont = win)
gbutton("cancel", cont=g)
gbutton("ok", cont=g)
```

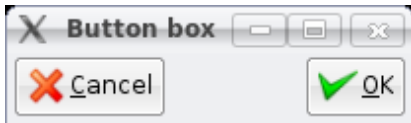


Button box with a "spring"

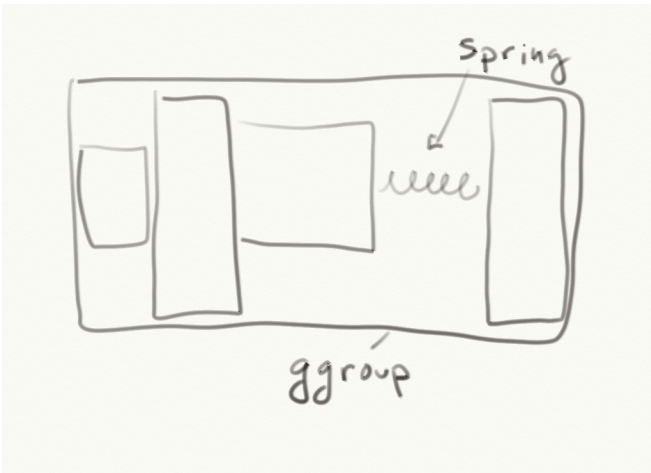
Use `addSpring` to push things to the right (or bottom)

Button box (take 2)

```
win = gwindow("Button box")
g = ggroup(cont = win)
gbutton("cancel", cont=g)
addSpring(g)
gbutton("ok", cont=g)
```



ggroup picture

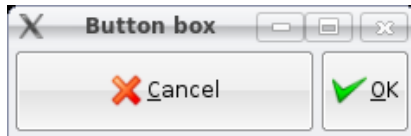


Button box (cont.)

The `expand=TRUE` argument will instruct the widget to fill as much space as it can.

Button box (take 3)

```
win = gwindow("Button box")
g = ggroup(cont = win)
gbutton("cancel", cont=g, expand=TRUE)
gbutton("ok", cont=g)
```



layout: a grid container

glayout

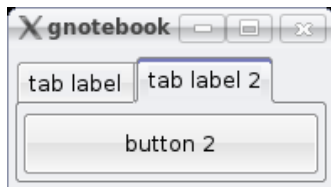
```
win = gwindow("glayout")
tbl = glayout(cont=win)
tbl[1,1] <- gbutton("1,1", cont=tbl)
tbl[1,2:3] <- gbutton("1,2:3", cont=tbl)
tbl[2:3,1:2] <- gbutton("2:3,1:2\n", cont=tbl)
tbl[2:3,3] <- gbutton("2:3,3", cont=tbl)
visible(tbl) <- TRUE ## RGtk2 only
```



gnotebook: For holding multiple pages

gnotebook

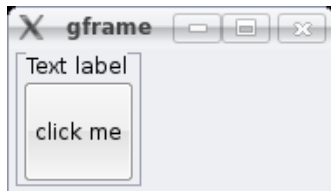
```
win = gwindow("gnotebook")
nb = gnotebook(cont=win)
gbutton("button", cont=nb, label = "tab label")
gbutton("button 2", cont=nb, label = "tab label 2")
```



gframe: A framed ggroup container

gframe

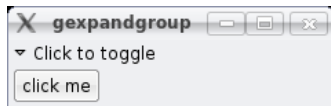
```
win = gwindow("gframe")
gp = ggroup(cont=win)
g = gframe("Text label", cont=gp)
gbutton("click me", cont=g)
```



gexpandgroup: Like a frame but body can be hidden

gexpandgroup

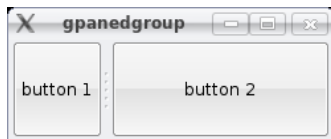
```
win = gwindow("gexpandgroup")
g = gexpandgroup("Click to toggle", cont=win)
gbutton("click me", cont=g)
visible(g) <- TRUE ## open up
```



gpanedgroup: adjustable pane between two groups

gpanedgroup

```
win = gwindow("gpanedgroup")
pg = gpanedgroup(cont=win)
gbutton("button 1", cont=pg)
gbutton("button 2", cont=pg) ## add twice
```



Container methods

Containers have a few new methods defined for them.

<code>add</code>	add a widget or container
<code>delete</code>	remove widget from container
<code>dispose</code>	destroy container
<code>visible<-</code>	set visibility of window

A simple GUI to control a plot

This example shows a simple dialog to control a plot by using a slider and an edit box. It could easily be extended. First, we define a function to make a plot of a histogram of m realizations of \bar{x}_n for a random sample of size n from the exponential distribution.

plotting function

```
makePlot = function(...) {  
  n = svalue(nWidget); m = svalue(mWidget)  
  x = matrix(rexp(n*m), nrow=n)  
  res = apply(x, 2, mean)  
  hist(res)  
}
```

The double use of `svalue` can be avoided, as shown in the next example.

Simple GUI (cont.)

Now we set up two widgets to set values for n and m . We use nested `ggroup` containers for a simple layout.

GUI layout

```
w = gwindow("Simple GUI")
g = ggroup(horizontal = FALSE, cont=w)
gp = ggroup(cont = g)
glabel("No. simulations (m)", cont=gp)
mWidget = gslider(from=10,to=250,by=10, cont=gp,
  expand=TRUE, handler=makePlot)
gp = ggroup(cont = g)
glabel("Size of sample (n)", cont=gp)
nWidget = gedit("5", cont=gp, coerce.with=as.numeric,
  handler=makePlot)
```

Simple GUI (cont.)

Finally, a button with a handler to call `makePlot` in case the GUI isn't clear.

Adding a button

```
gbutton("New sample", cont=g, handler = makePlot)
```

A new graph is produced when the slider or test widget are changed (the latter when the enter key is pressed), or when the button is clicked.

A more ambitious GUI for qplot (ggplot2)

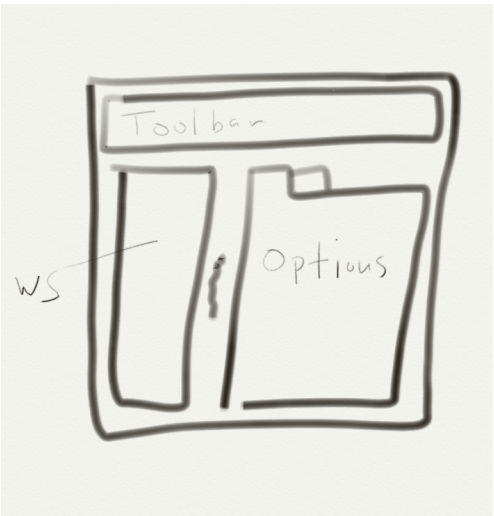
The goal of this is to show some of the steps needed to make a GUI ¹ for the `qplot` function (quick plot) which is part of the `ggplot2` package. This function has many arguments:

```
> args(qplot)
function (x, y = NULL, z = NULL, ..., data, facets = . ~ . ,
  margins = FALSE,
  geom = "point", stat = list(NULL), position = list(NULL),
  xlim = c(NA, NA), ylim = c(NA, NA), log = "", main = NULL,
  xlab = deparse(substitute(x)), ylab = deparse(substitute(y)),
  add = NULL)
```

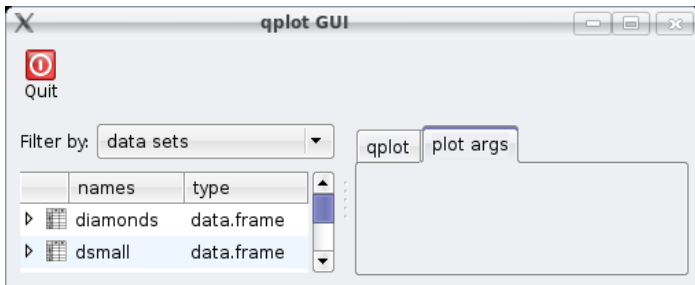
First, a “sketch” of what the GUI layout is to be:

¹The code for this example is at

qplot GUI application sketch



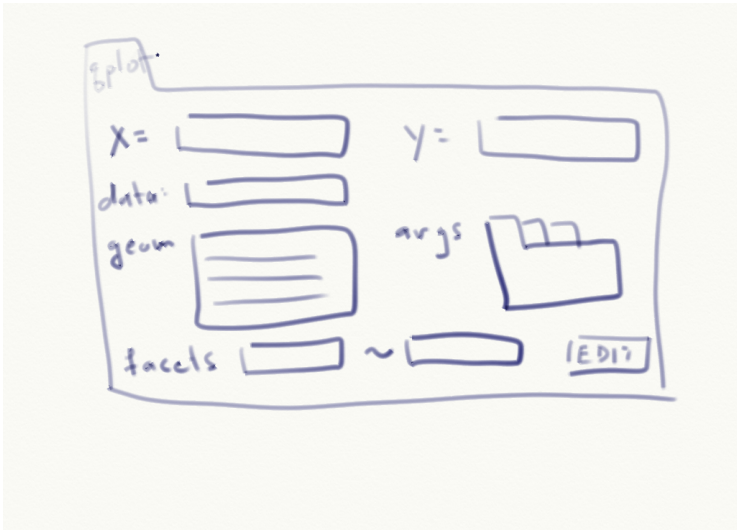
qplot GUI application laid out



GUI Layout code

```
win = gwindow("qplot GUI")
g = ggroup(horizontal=FALSE, cont=win, expand=TRUE)
## toolbar
tbl = list(Quit=list(handler=function(h,...) dispose(win), i
tb = gtoolbar(tbl, cont=g)
## paned group
pg = gpanedgroup(cont=g, expand=TRUE)
vb = gvarbrowser(cont=pg) ## workspace browser
nb = gnotebook(cont=pg) ## notebook
qpg = ggroup(horizontal=FALSE, cont=nb, label="qplot")
parg = ggroup(horizontal=FALSE, cont=nb, label="plot args")
```

Qplot argument tab sketch (cont.)



Some code to make this

```
geomVals1d = c("histogram","density")
tbl = glayout(cont=qpg)
tbl[1,1, anchor=c(1,0)] <- "x"
tbl[1,2] <- (widgets[['x']] <- gdroplist(c(),editable=TRUE,
tbl[1,3, anchor=c(1,0)] <- "y"
tbl[1,4] <- (widgets[['y']] <- gdroplist(c(),editable=TRUE,
tbl[2,1, anchor=c(1,0)] <- "data"
tbl[2,2] <- (widgets[['data']] <- gedit("", cont=tbl))
tbl[2,3, anchor=c(1,0)] <- "weights"
tbl[2,4] <- (widgets[['weights']] <- gdroplist(c(),editable=
tbl[3,1:4] <- gseparator(cont=tbl)
tbl[4,1, anchor=c(1,0)] <- "geom"
tbl[4:8,2] <- (widgets[['geom']] <- gtable(geomVals1d,multip
### ... more goes here ....
visible(tbl) <- TRUE
```


qplot argument layout (actual one)

The screenshot shows a window titled "qplot GUI argument collector" with standard window controls (minimize, maximize, close). The interface is organized into several sections:

- Top row:** Two dropdown menus labeled "x" and "y".
- Second row:** A text input field labeled "data" and a dropdown menu labeled "weights".
- Third row:** A list box labeled "geom" containing "items", "histogram", and "density". To its right is a text input field labeled "args".
- Bottom row:** Two dropdown menus labeled "facets" (with a "." in the first) and another "facets" (with a "." in the first), separated by a tilde "~". To the right is an "Edit" button with a pencil icon.
- Bottom-most row:** A text input field labeled "add".

Key handlers

The desired *event handlers* are

- Drag and drop data frame from variable browser onto data widget: **update x, y, facets droplists.**
- select an x and a y value: **update possible geoms to be 2d**
- Select a geom: **if there are any args, fill into args area**
- Change a value: **produce a plot if possible**
- Click on “Edit”: **edit formula for facets** (done automatically)

Handler code snippets

Update variable names

```
sapply(c("x", "y", "weights"),
      function(i) widgets[[i]][ ] <- c("", theNames))
```

change geoms

```
if(svalue(widgets[['y']]) == "")
  widgets[['geom']][, ] <- geomVals1d
else
  widgets[['geom']][, ] <- geomVals2d
```

Fill in args area (trickiest part) This is a notebook. Need to remove any extraneous arguments then add in any new ones.

Produce a plot. Basic idea is to collect arguments with

```
tmp = lapply(widgets, svalue) ## all outputs
l = list()                    # store args here
if(tmp$data != "")           ## there is a data frame
## ... fill in l more then
print(do.call("qplot", l))
```

Bind handlers to widgets

update graphic

```
sapply(names(widgets), function(i) {  
  addHandlerChanged(widgets[[i]], handler = function(h,...)  
    updateGraphic()  
  })  
})
```

Click handler for geom (double click is "changed")

```
addHandlerClicked(widgets[['geom']], handler = updateGraphic)
```

Adding data Needs a "changed" handler *and* a drop target

```
addHandlerChanged(widgets[['data']], handler=function(h,...)  
  updateVarNames(svalue(h$obj))  
  updateGraphic()  
})  
addDropTarget(widgets[['data']], handler = function(h,...) {  
  updateVarNames(h$dropdata)  
  updateGraphic()  
})
```

qplotGUI screenshot

