0.1 Introduction

There are many reasons why one would want to create a graphical user interface for a package or piece of R functionality. For example,

- GUIs allow others easier access to your functions,
- GUIs are useful if a function has many arguments,
- GUIs can be dynamic, they can direct the user how to fill in arguments, can give feedback on the choice of an argument, they can prevent or allow user input as appropriate,
- GUIs make dealing with large data sets easier both visually and as an alternative to sometimes difficulte command line usage.
- GUIs can tightly integrate with graphics, for example the rggobi interface among others.

Even though R has so much new interest, especially commericially supported interest it does not have a common graphical user interface. Many reasons exists. Historically those most capable of doing so in the R community had the least desire for one, the cross-platform nature of R makes it difficult to provide, the wide variety of user types lends a single interface unlikely to satisfy all. Even if it did have a common interfface, as much of R's functionality depends on user contributions there will always be a demand for package programmers to provide convenient interfaces to their work.

0.2 A simple GUI in R

We begin with an example showing how one can use R's standard graphics device, as a canvas for drawing a GUI, for playing a game of tic-tac-toe against the computer (Figure **??**). Although this example has nothing to do with statistics, it illustrates, in a familiar way, some of the issues that this text will address with using GUIs.

Many GUIs can be thought of as different views of some data model. This thought leads itself to the prevalent model-view-controller architecture of many large GUI applications. For many our the examples in the text, we avoid that level of design, but do try to keep separate the different aspects of GUI design. In this example, the data simply consists of information holding the state of the game. Here we define a global variable, `board`, to store the current state of the game.

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

The GUI provides the representation of the data for the user. This example just uses a canvas for this, but most GUIs have a combination of components to represent the data and allow for user interaction. The layout of the GUI directs the user as to how to interact with it and is an important factor as to whether the GUI will be well recieved. Here we define a function to layout the game board using the graphics device as a canvas.

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

A GUI is designed to respond to user input typically by the mouse or keyboard. The underlying toolkit allows the programmer to assign functions to be called when some specific event occurs. Typically, the toolkit *signals* that some action has occurred, and then calls *callbacks* or *event handlers* that have been assigned by the programmer. How this is implemented varies from toolkit to toolkit.

R's interactive graphics devices implement the `locator` function which responds to mouse clicks by user. When using this function, one specifies how many mouse clicks to gather and the *control* of the program is suspended until these are gathered (or the process is terminated). The suspension of control makes this a *modal* GUI. This design is common for simple dialogs that require immediate user attention, but not common otherwise. To make non-modal dialogs possible, the writers of the R packages that interface with the GUI toolkits have to interface with R's event loop mechanism.

Here we define a function to call to collect the player's input.

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

In the above fuction, `clickHandler` is an *event handler*. Its job is to process the output of the `locator` function, checking first if the user terminated `locator` using the keyboard. If not it proceeds to draw the move, and then, if necessary, the computer's move. Afterwards, play is repeated until there is a winner or a "cat's" game.

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

The use of `<<-` in the handler illustrates a typical issue in GUI designing within R. After a GUI is created, the state is typically modified within the

scope of the callback functions. These callbacks need to be able to modify values outside of their scope, yet even if the values are passed in as argument, this is usually not possible while assigning within the scope of the function call, due to R's pass by copy function calls. As such, global variables are often employed along with some strategies to avoid an explosion of variable names.

Validation of user input is an important task for a GUI, especially for Web GUIS. In the above, the `clickHandler` function checks to see if the user terminated the game early and issues a message, more helpful forms of validation are possible in general.

At this point, we have a data model, a view of the model and the logic connects the two, but we still need to implement some of the functions to tie it together.

This function draws either an "x" or an "o". It does so using the `lines` function. The `z` argument contains the coordinates of the square to draw.

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

One could use `text` to place a text "x" or "o", but this may not look good if the GUI is resized. Most GUI layouts allow for dynamic resizing. Overall this is a great advantage, for example it allows translations to just worry about the text and not the layout even though some languages are much more verbose and hence require more space.

This function is used to test if a game is finished. The matrix `m` allows us to easily check all the possible ways to get three in a row.

```
isFinished <- function() {
  if(sum(abs(board)) >= 9)
    return(TRUE)
  m <- matrix(1:9,nrow=3)
  ways <- list(m[,1], m[,2], m[,3],
               m[1,], m[2,], m[3,],
               diag(m),diag(apply(m,2,rev)))
  sums <- sapply(ways, function(i) abs(sum(board[i])))
  if(any(sums == 3))
    return(TRUE)
  return(FALSE)
}
```

This function picks a move for the computer. The move is converted into coordinates using %% to get the remainder and %/% to get the quotient when dividing an integer by an integer. This function just chooses at random from the left over positions, we leave implementing a better stategy to the interested reader.

```
doComputerMove <- function() {
  ## random !
  newMove <- sample(which(board == 0),1)
  board[newMove] <<- -1                  # otherwise a copy of board
  z <- c((newMove-1) %% 3, (newMove-1) %/% 3) + 1
  drawMove(z,"o")
}
```

This function provides the main entry point for our GUI. To play a game it first lays out the board and then calls `doPlay`. When this function terminates, a message is written on the screen.

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

Finally, the above example illustrates a common issue when designing GUIs – they seemingly can always be improved. In this case, there are many obvious improvements: localizing the text messages so different languages can be used, implementing a better logic for the computer's moves, drawing a line connecting three in a row when there is a win, indicating who won when there is a win, etc. For many GUIs there is a necessary trade-off between offering sufficient usability versus increased complexity.

0.3 GUI Design Principles

A typical interface design consists of a top-level window referred to as the *document window* that shows the current state of a "document", whatever that is taken to be. In R it could be a data frame, a command line, a function editor, a graphic etc.. The actions that can be done on the "document" are called through the menubar or toolbar and have their parameters controlled in sub-windows. These sub-windows are termed *application windows* by Apple (**?**), but we prefer the term *dialogs*, or *dialog boxes*. These terms may also refer to smaller sub-windows that are used for alerts or confirmation. As these are often used in a modal manner, we refer to them as *modal dialog boxes*.

Each window or dialog typically consists of numerous widgets or controls layed out in some manner to facilitate the user interaction. However, there are numerous means to achieve the same goals. For example, Figure 0.1 contains three dialogs that perform the same task – collect arguments from the user

to customize the printing of a document. Although all were designed to do the same thing, there are many differences.

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

How the interfaces are layed out also varies. All are read top to bottom, although the Mac interface also has a very nice preview feature on the left side. The firefox and KDE dialogs use frames to separate out the printer arguments from the arguments that specify how the print job is to proceed. The Mac uses a vertical arrangement to guide the user through this. For the Mac, horizontal separators are used instead of frames to break up the areas, although a frame is used towards the bottom.

Apple uses a center balance for its controls. They are not left justified as are the KDE and firefox dialogs. Apple has strict user-interface guidelines and this center balance is a design decision.

The Mac GUI provides a very nice preview of the current document indicating to the user clearly what is to be printed and how much. Adjusting GUIs to the possible state is an important user interface property. In each GUI areas that are not currently sensitive to user input are grayed out. For example, the "collate" feature of the GUI only makes sense when more than 1 copy is selected, so the designers have it grayed out until then. It is common practice when designing GUIs to only enable controls that initiate actions when those actions can actually be completed given the state of the application.

The Mac GUI has the number of pages in focus, whereas for the firefox one the printer is focused. This allows the user to interact with the GUI without the mouse. Typically the tab key is used to step through the contorls. GUIss often have keyboard accelerators, such as the KDE dialog where the underlined letters indicate the accelerator. Such additions help power users navigate through the interface quicker. As well most dialogs have a default button, which will initiate the action when clicked on or if the return key is pressed. In the KDE dialog, you can see by the shading that this is the "print" button.

For such a common dialog, it is unlikely the user will need help. As such the firefox dialog does not provide a link. However, the KDE and Mac dialogs do. A typical dialog should if the functions it provides are not commonly carried out.

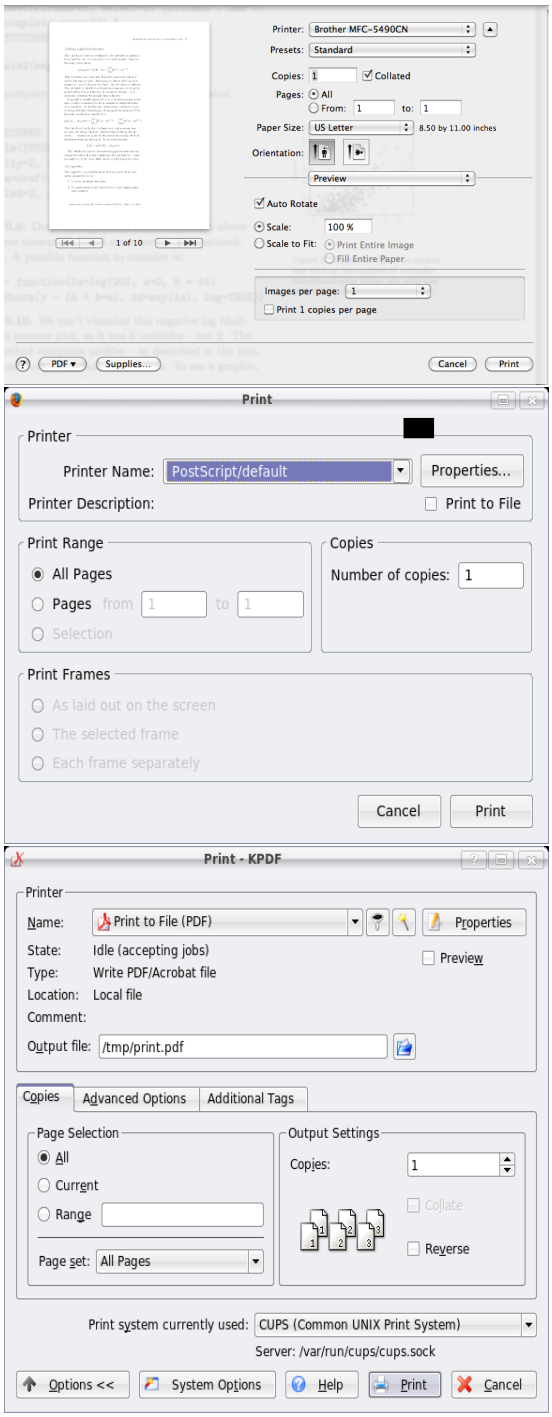

Figure 0.1: Three print dialogs. One from Mac OS X 10.6, one from firefox 2.0 and one from KDE 3.5.

Table 0.1: Table of possible selection widgets by data type and size

Type of data	Single	Multiple
Boolean	Checkbox	
Small list	radiogroup	checkboxgroup
	combobox	list box
	list box	
Moderate list	combobox	list box
	list box	
Large list	list box	list box
Sequential	slider spinbutton	
Tabular	table	table
Heirarchical	tree	tree

The Apple human interface guidelines suggest putting buttons that can cause the destruction of data separate from other control buttons. As this isn't directly applicable here, we see that Apple does separate buttons that are common to many dialogs (cancel, print) from the ones specific to the dialog. The KDE buttons have nice icons, but their similare, but irregular, sizing is a bit unusual.

One of the greatest differences between these dialogs is the amount of features they choose to expose. The firefox dialog is essentially minimal by comparison. As this is for printing from a browser, and not other documents it makes sense. The Mac GUI uses "disclosure buttons" to allow access to printer properties and the PDF settings, whereas KDE uses a notebook container to show possible options without revealing all their detail.

Choice of widget

A GUI is comprised of one or more widgets. Typically these widgets allow a user to make a selection from one or more possible values. The choice of widget varies greatly depending on the length and type of values, on the desired number to select, and on the pixel-size allowable in the GUI layout. Table 0.3 lists several different types of widgets used for selection from a set of values.

Figure 0.2 shows several such controls in a single GUI. A checkbox is used to include an intercept, a radio group is used to select either full factorial or a custom model, a combobox is used to select the "sum of squares" type, and a list box is used to allow for multiple selection from the available variables in the data set.

For many R object types there are natural choices of widget. For example, values from a sequence map naturally to a slider or spin button; a data frame maps naturally to a table widget; or a list with similar structure can

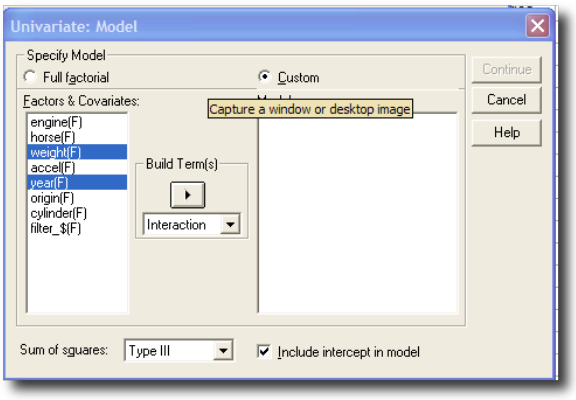

Figure 0.2: A dialog box from SPSS version 11 for specifying terms for a linear model. The graphic shows a dialog that allows the user to specify individual terms in the model using several types of widgets for selection of values, such as a radio group, a checkbox, combo boxes, and list boxes.

map naturally to a tree widget. However, certain R types have less common metaphors. For instance, a formula object can be a fairly complex thing. Figure 0.2 shows an SPSS dialog to build terms in a model. R power users may be much faster specifying the formula through a text entry box, but beginning R users coming to grips with the concept of a command line and the concept of a formula may benefit from the assistance a well designed GUI can provide. Designing an interface to balance both of these types of users may be desired, or one may decide to use an interface such as the cumbersome SPSS one. Knowing the potential user base is important.

0.4 Controls

This section provides an overview of many common GUI controls.

Selection

A major taks for a GUI for statistics is the selection of a value. For R there are conceivably several different possible types of selections. For example selecting a data frame from a list of data frames, selecting a variable in a data frame, selecting certain cases in a data frame, selecting a logical value for a function argument, selecting a numeric value for a confidence level, selecting a string to specify an alternative hypothesis. Clearly there can be no one-size-fits all widget to handle the selection of a value. We describe some standard selection widgets next.

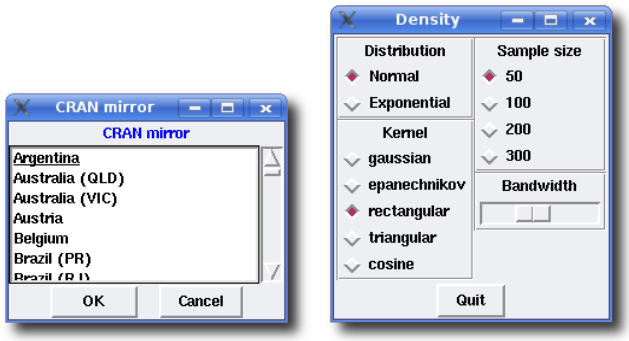

Figure 0.3: Two applications of the `tcltk` package. The left graphic is produced by `chooseCRANmirror` and uses a list box to allow selection from a long list of possibiulities. The right graphic is the `tkdensity` demo from the package. It has various widgets that allow for the selection of parameter values for a density plot.

Checkboxes

A *checkbox* allows the user to select a logical value for a variable. Checkboxes have labels to indicate which variable is being selected. A *checkbox group* is a linked collection of checkboxes that allows the selection of one or more values at a time.

Radio Button Groups

A *radio button group* allows a user to select exactly one value from a vector of possible values. The analogy dates back to old car radios where there were 6 buttons to press to select a channel. When a new button was pushed in, the old button popped up. Radio button groups are useful, provided there are not too many values to choose from, as all the values are shown. These values can be arranged in a row, a column or rows and columns to better use screen space.

Sliders and spinbuttons

A *slider* is a widget that allows the selection of a value from a sequence of possible values (typically) through the manipulation of a knob that can visually range over the possible values. Some toolkits (e.g. Java) only allow for the sequence to have integer values. The slider is a good choice for offering the user a selection of parameter values. The `tkdensity` demo of the `tcltk` package (Figure 0.3) uses a slider to dynamically adjust the bandwith of a density estimate.

A *spin button* also allows the user to specify a value from a possible sequence of values. Typically, this widget is is drawn with a text box displaying the current value and two arrows to increment or decrement the selection. The text box can usually be edited. Some toolkits allow the sequence to be a more general than a numeric sequence. A spin button has the advantage of using less screen space, but is much harder for the user to use if the sequence to choose from is long. A spin button is used in the KDE print dialog of Figure 0.1 to adjust the number of copies.

Popup menus and Combo boxes

A *popup menu* is a widget that allows the selection of one or more fixed values while only showing just the currently selected one. Usually an arrow is drawn beside the selected value to indicate to the user a choice is possible. Clicking on this arrow will cause a pre-specified list of values to either drop down or up depending on the location of the widget on the screen. If there are too many choices, then typically further arrows are used to indicate more available choices. From a screen-space perspective, they can efficiently allow the selection of a value from many values, although a choice from too many values can be annoying to the user, as anyone who has had to select their country out of over 100 in a web form can attest. Some toolkits, such as GTK+ allow the specification of an icon next to the list of values. A *combo box* is a popup menu combined with a text entry area to allow the user to specify their own value.

List boxes

A *list box* is a widget that displays in a column the list of possible choices. Often a scrollbar is used when the list is too long to show in the allocated space. Some toolkits have list boxes that allow the values to spread out over several columns. Selection typically occurs with a right mouse click or through the keyboad, whereas a double-click will typically be programmed to initiate some action. Figure 0.3 shows a list box created by R that is called from the function `chooseCRANmirror`.

List boxes are a good alternative to drop lists when the number of selections gets bigger thatn 30. Additionally, list boxes can be programmed to allow for multiple selection. This is typically done by holding down either the shift or ctrl keys.

Some toolkits allow widgets to be placed next to the entries, such as checkboxes or icons. The right-most graphic in Figure **??** shows how SPSS places an icon indicating the type of variable next to the variable name in a list box, to aid in selection of the proper type of variable.

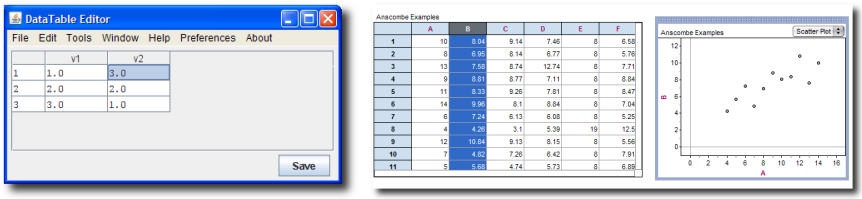

Figure 0.4: Two windows showing the use of table widgets. The left graphic shows the data editor from `JGR` using the table widget in Java. The right graphic shows a data table and a graph in Fathom 2.1 with two views of the same data. One view uses a table widget, the other a graph. Changes to one or the other views cause an update to the underlying model. This model then will notify its various displays to update. This arrangement allows for dynamic linking of the table and the graph.

Tabular display

A *table widget* shows tabular data, such as a data frame, where typically each column is rendered in the same manner but different columns need not be. The table widget need not be as flexible as a spread sheet, despite their similar appearances. For example, in GTK+ the selection is done by row, as in a list box.

Table widgets may be used to display and select tabular data, or may be intended for the user to also edit the data. Figure 0.4 shows the data editor from the `JGR` GUI that uses Java's table widget.

In `RGtk2` and `rJava`, the table widget has a separation of the data (the model) and its display. This has many advantages, such as, allowing for sorting and filtering of the values displayed without changing the underlying data, allowing for multiple views of the same data; or allowing for efficient ways to store, perhaps, sparsely filled tables. Figure 0.4 shows a table object and a graph object in a Fathom session. These are two different views for the same data model. The two are dynamically linked, as changes to the underlying model are propogated to its views so that they are updated.

The view part of a table, for efficiency, is geared around the display of its columns rather than on a cell-by-cell basis (although this may be overridden, such as when a cell is being edited). The programmer then needs to specify a mapping of data in the model to column numbers in the display and the means of rendering the data in the column.

Tree widgets

List boxes essentially display vectors of similar data, essentially tables display data frames. If the data has a heirarchical structure, then a *tree widget* can be used to show that. Examples of heirarchical data in R, are directory

structures, the components of a list, or the inheritance of methods. The object browser in JGR uses a tree widget to show the components of the objects in a users session (the left graphic of Figure 0.5). The root node of the tree is the "data" folder, each data object in the global workspace is treated as an offspring of this root node. For the data frame iraq, its variables are considered as offspring of the data frame. In this case these variables have no further offspring, as indicated by the "page" icon.

Inititiating an action

When users use some widgets, they expect some immediate action to occur. Examples of these widgets are the familiar buttons, menubars and toolbars.

Buttons

A *button* is typically used to give the user a place to click the mouse in order to initiate some immediate action. The "Properties" button, when clicked, causes a dialog for setting a printers properties to open. Similarly, the button with the wizard icons does something similar. As buttons typically lead to an action, they often are labeled with a verb. (**?**) In Figure 0.2 we see how SPSS uses buttons in its dialogs: buttons which are not valid in the current state, as the user has not input enough information are disabled; buttons which are designed to open subsequent dialogs have trailing dots; and the standard actions of resetting the data, canceling the dialog or requesting help are given their own buttons on the right edge of the dialog box.

The look of the button can usually be manipulated. A button is given a relief through its border, shading, and perhaps a color gradient along its face. Some toolkits allow these to be optionally drawn, thereby making a button look more like a label, as described below. The button text may have some markup or an indication of a accelerator keyboard binding, such as the Contrasts... button in the dialog shown in the right graphic of Figure **??**.

icons

An *icon* is used to augment or replace a text label on a button, a toolbar, in a list box, etc.. When icons are used on toolbars and buttons, they are associated with actions, so the icons should have some visual association. Well drawn icons make a big visual difference in a GUI. The KDE print dialog, contrasted to the firefox print dialog in Figure 0.1 illustrates this point.

Except for the default installation of tcltk, images and icons may be specified in a variety of different formats. Icons can come in several different sizes from 16 by 16 pixels to 128 by 128. For toolbars and menubars, the toolkit takes care of selecting the appropriate icon.

Menubars

Xerox Parc's revolutionary idea of a `WIMP` GUI added windows, icons, menubars, and pointing devices to the desktop computing environment. The central role of menu bars has not diminished. For a GUI, the *menubar* gives access to the full range of functionality available. Each possible action should have a corresponding menu item. Menubars are traditionally associated with a top-level window. This is enforced by the toolkit in wxWidgets and Java but not Tcl/Tk and GTK+. In Mac OS X, the menubar appears on the top line of the display, but otherwise they typically appear at the top of the main window. In most modern applications, standard document-based design is used to organize a GUI and its actions, with a main window showing the document and its menu bar calling actions on the document, some of which may need to open subsequent application windows or dialogs for gathering additional user input. In this model, only the main window has a menu bar not the application windows or dialogs. In a statistics application, the "document" may be viewed as the active data frame, a report, or a graphic so there may be many different menu bars needed.

The styles used for menubars are fairly standardized, as this allows new users to quickly orient themselves with a GUI. The visible menu names are often in the order `File`, `Edit`, `View`, `Go`, then application specific menus, and finally a `Help` menu. Each visible menu item when clicked opens a menu of possible actions. The text for these actions traditionally use a `...` to indicate that a subsequent dialog will open so that more information can be gather to complete the action, as opposed to some immediate action being taken. The text may also indicate a key-board accelerator, such as `Find Next F3` indicating that both "N" as a keyboard accelerator and F3 as a shortcut will initiate this same action.

Not all actions will be applicable at any given time. It is recommended that rather than deleting these menu items, they be disabled, or greyed out, instead. **??**

Menus can get very long. To help the user the menu items are usually grouped together, first by being under the appropriate menu title, then with either horizontal separators to define subsequent groupings, or hierarchical submenus. The latter are indicated with an arrow. Several different levels are possible, but navigating through to many can be tedious.

The use of menus has evolved to also allow the user to set properties or attributes of current state of the GUI. There may be checkboxes drawn next to the menu item or some icon indicating the current state.

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

Toolbars

Toolbars are used to give immediate access to the frequently used actions defined in the menubar. Toolbars typically have icons representing the action and perhaps accompanying text. They traditionally appear on the top of a window, but sometimes are used along the edges.

Displaying text for editing

As much as possible, GUIs are designed around using the pointing device to select value for user input. Perhaps this is because it is difficult to both type and move the mouse at the same time. For statistical GUIs, especially for R with its poweful command line, it is essential that text entry widgets be used for any complicated GUI. Except with wxWidgets there is a distinction made between widgets for handling just single lines versus multiple lines of text.

Single line text

A text entry widget for editing a single line of text is used in the Firefox print dialogs in Figure 0.1 to specify the number of copies. An advantage to using a text entry area for this specification, as opposed to a spinbutton, is the ease of editing for a user who uses the tab key to move between fields rather than the mouse. A disadvantage of using this type of widget is the need to validate the user's input, as it only makes sense to have a positive integer for this value.

Text edit boxes

Multi-line text entry areas are used in many GUIs. The right graphic of Figure 0.5 shows a text entry areas used by `Rcmdr` to enter in R commands; to show the output of R commands and to provide a message area (in lieu of a status bar). The "Output Window" shows that text may have formatting attributes applied to it, in this case the color of the commands differs from that of the output. This is done by specifying text-changing attributes to different portions of the text. The difficulty is that when the text widget is changed, these positions must also reflect the change. Alternately, some of the toolkits allow HTML to be used for markup.

Display of information

Some widgets are typically used to just display information and are not expected to have any response to mouse clicks. These are called static controls in wxWidgets.

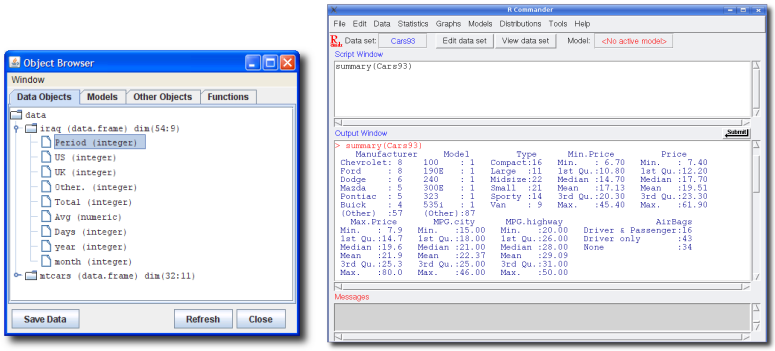

Figure 0.5: Some windows from R GUIs. The left graphic shows the object browser in the `JGR` GUI using a tree widget to display the possibly heirarchical nature of R objects. The right graphic shows the main Rcmdr (1.3-11) window illustrating the use of multi-line text entry areas for a command area, an output area and a message area.

Labels

A label is a widget for placing text into a GUI that is typically not intended for editing, or even selecting with a mouse. This widget is used to label other controls, so the user understands what will happen when that control is changed.

A Label's text can be marked up in some toolkits using HTML in Java/Swing and pango markup in GTK+. As well, regular font attributes can be applied to the label in the other toolkits. Figure **??** shows labels marked in red and blue in `tcltk`.

Statusbars

A typical top-level window will have a menubar and toolbar for access to the possible actions, an area to display the document being worked on, and at the bottom of the window a statusbar for giving the user immediated feedback on the actions that have been intiated. Figure **??** show a text area being used as a status bar for messages in the `Rcmdr` GUI. Likely this was chosen as there is no ready-made status bar widget in the base `tcltk` package. The other toolkits do provide one.

Progress bars

A progress bar is used to notify how much time is left during a long computation. They are often used during software installation.

Tooltips

A tooltip is a small window that pops up when a user hovers their mouse over a tooltip-enabled widget. These are used to give quick feedback on that particular widget. They can be useful for guiding new users of a GUI and annoying for familiar users. In R their implementation for the toolkits can be hampered due to eventloop issues.

Modal dialog boxes

A *modal dialog box* is a dialog box that keeps the focus until the user takes an action to dismiss the box. They are used to notify the user of some action, perhaps asking for confirmation in case the action is destructive, such as overwriting of a file name. Modal dialog boxes can be disruptive to a users flow, so should be used sparingly. As the flow essentially stops until the window is dismissed, functions that call modal dialogs can return a value when an event occurs, rather than have a handler respond to the dismiss event. The `file.choose` function, mentioned below, is a good example. When used during an interactive R session, the user is unable to interact with the command line until a file has been specified of the dialog dismissed.

File choosers

A file chooser allows for the selection of existing files, existing directories, or the specification of new files. They are familiar to any user of a GUI. A typical R installation has the functions `file.choose` and `tkchooseDirectory` (in the `tcltk` package) to select files and directories, the latter using `tcltk` and the former using the windows file selector if applicable. The use of

Other common choosers are color choosers and font choosers.

Message dialogs

A *message dialog* generally has a pretty standard form, a small rectangular box that appears in the middle of the screen with an icon on the left and a message on the right. At the bottom is a button to dismiss the dialog, often labeled "OK." A *confirmation dialog* would add a "cancel" button which the programmer can use to invalidate the proposed action.

0.5 Containers

Widgets are arranged in a window to produce a GUI. How they are laid out is done using containers. The simplest containers a like boxes that get packed in left to right or top to bottom. These boxes may be decorated with a frame or label, or may have some means of being hidden or displayed by the user. The nesting of box containers can provide a great deal of flexibility, but

usually not enough. Additionally, layout can be done using grids to specify where to place the widgets.

Top level windows

The top-level window of a GUI or its sub-windows typically is drawn with window decorations including a title and buttons to iconify, maximize, or close. In addition, a top-level window may have room for a menu bar, a tool bar and a status bar. In between these, if present, is the area referred to as the *client area* or *content pane* where other containers or widgets are placed.

The title is a property of the window and may be specified at the time of construction or afterwards.

On a desktop, only one window may have the focus at a time. It may or may not be desired that a new window receive the focus so some means to specify the focus at construction or later is provided by the toolkit.

The initial placement of the window also may be specified at the time of construction. The top-level window of a GUI may generally be placed whereever it is convenient for the user, but sub-windows are often drawn on top of their parent window, as are modal dialog boxes.

The initial size of the window may also be specified at construction. The size of a top-level window may be specified as a default size, a preferred size, or a minimum size, depending on the toolkit. All of these allow for resizing of the window with the mouse. When this is done, the laying out of widgets must be updated.

The window manager usually decorates a top-level window with a close button. It may be necessary or desirable to specify some action to take place when this button is clicked. For instance, a user might be prompted if they wish to save changes to their work when the close button is pressed.

it may take some time to initially layout a top-level window. Rather than have the window drawn and then have a blank window while this time passes, it is preferable to not make the window visible until the window is ready to draw.

The layout of widgets in a top-level container follows a heirarchical structure. Widgets are packed inside containers which may again be packed inside containers. Eventually the widget or container is packed inside a top-level window. In `tcltk` a parent container must be specified when a widget is constructed. Reversing the direction, one can view the top-level window as a root node of a tree with each container packed directly into it as children. (In `RGtk2` only one component can be packed into a top-level window.) Each child container, in turn, has children which are the components packed into it. This heirarchy is important when a window is resized. The new dimensions can be passed along to the children, and they can resize themselves accordingly. The allocation of this space varies from toolkit to toolkit.

We now describe some of the main containers.

Box containers

A box container places its children in it from left to right or from top to bottom. If each child is viewed as a box, then this container holds them by packing them next to each other. The upper left figure in Figure0.6 illustrates this.

When the boxes have different sizes, then some means to align them must be decided on. Several possibilities exists. The alignment could be around some center, aligned at a baseline, the top line, or each child can specify where to anchor itself within the allotted space (the upper right graphic in Figure 0.6).

If the space allocated for a box is larger than the space requested by a child component then a decision as to how that component gets placed needs to be made. If the component is not enlarged, then there are nine reasonable places – the center and the 8 compass directions N, NW, W, ... Otherwise, it may be desirable for the component to expand horizontally, vertically or both (the lower left graphic in Figure 0.6). Additionally, it is desirable to be able to place a fixed amount of space between child components or a spring between components. A spring forces all subsequent to children to the far right or bottom of the container (the lower right graphic in Figure 0.6).

When a top-level window is resized, these space allocations must be made. To help, the different toolkits allows the components to have a size specifed. Some combination of a minimum size, a preferred size, a default size, a specific size, or a maximum size are allowed. Specifying fixed sizes for components is generally frowned upon, as they don't scale well when a user resizes a window and they don't work well when different languages are used on the controls when an application is localized.

Frames

A box container may have a border drawn around it to visually separate its contents from others. This border may also have a title. In GTK+ these are called frames, but this word is reserved in Java and wxWidgets for windows.

Expanding boxes

In order to save screen space, a means to hide a boxes contents can be used. This hiding/showing is initiated by a mouse click on a button or label.

Paned boxes

If automatic space allocation between two child components is not desired, but rather a means for the user to allocate the space is, then a *paned container*

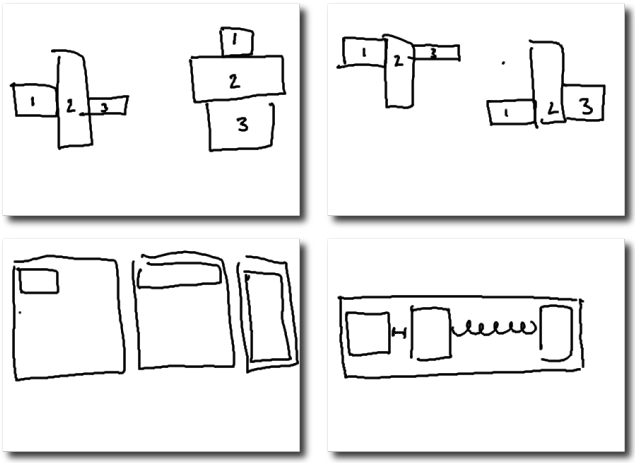

Figure 0.6: XXX REPLACE WITH REAL GRAPHICS XXX. Different possibilities for packing child components within a box. The upper left shows horizontal and vertical layout. The upper right shows some possible alignements. The lower left shows that a child could possibly be anchored in one of 9 positions. As well, it could "expand" to fill the space either horizontally (as shown) or vertically, or both. The lower right shows both a fixed amount of space between the children and an expanding spring between the child components.

may be used. These offer either a horizontal or vertical sash that can be clicked and dragged to apportion space between two components.

Grid layout

By nesting box containers, a great deal of flexibility in layout can be achieved. However, there is still a need for the alignment of child components in a tabular manner. The most flexible alignments allow for different sizes for each column and each row, and additionally, the ability for the child components to span multiple columns or rows. Within each cell (or cells) the placement of a child component mirrors that of the lower left graphic of Figure 0.6. Some specification where to anchor the component when there are nine possible positions plus expanding options must made.

Notebooks

A notebook is a common container to hold one or more pages (or children). The different pages are shown by the user through the clicking of a corresponding tab. The metaphor being a tabbed notebook. Modern web browsers

take advantage of this container to allow several web pages to be open at once.

Example

The KDE pring dialog of Figure 0.1 shows most of the containers previously described.

The top-level window has the generic title "Print – KPDF." This window appears to have four child components: a frame labeled `Printer`, a notebook with open tab `Copies`, a grid layout for specifying the print system, and a box for holding five buttons at the bottom.

The lower left `Options` button has << to indicate that clicking this will close an exanding box, in this case a box that contains the lower three components above. So in fact, there are two visible child components of the top-level window.

The framed box holds a grid layout with five columns and 6 rows. The sizes allocated to each column are visible in the first row. It is quickly seen that each column has a different size. The last row has a text entry area that spans the second and third columns. The first column has only labels. These are anchored to the left side of the allowed space. The Apple human interface guide (**?**, p. 124) suggests using colons for text that provides context for controls, and the KDE designers do to.

The displayed page of the notebook shows a two child components, both framed boxes. A pleasant amount of space between the frames and their child components has been chosen. The `Page Selection` frame has components including radio buttons, a text area, a horizontal separator, and a combo box.

The print system information is displayed in a grid layout that has been right aligned within its parent container – the expanded group, but its children are center-balanced with the label "Print system currently used" is right aligned and "Server..." is left aligned within their cells.

The button box shows five buttons as child components. At first glance the sizing appears to show that each button is drawn to fully show its label with some fixed space placed between the buttons. If the dialog is expanded, it is seen that there is a spring between the 3rd and 4th buttons, so that the first 3 are aligned with the left side of the window and the last two the right side.

The Apple guidlines(**?**, Ch. 15) suggest using "center equalization" for arranging widgets within a window. This means that the visual weight is balanced between the right and left side of the content area. This is not the case with the KDE print dialog.

0.6 End of chapter notes

More documentation on GUIs is available in book format or online.

For GTK+ there is the gtk tutorial (pygtk); GTK API; DTL's notes; example code in the `RGtk2` package; php-gtk cookbook

For wxWidgetsthe book; DTL omegahat pages; wxWidgets API;

For Tcl/Tk ActiveStates API; wettenhall examples (sciviews); Dalgaard's papers; R mailing list; book

For Java Sun's website tutorials; API; rJava package page;

Event loops