

# Building a widget using widgetTools

Jianhua Zhang

May 12, 2008

©2002 Bioconductor

\$Id: widgetTools.Rnw 6285 2003-10-23 13:26:15Z jhnzhang \$

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Build a widget</b>	<b>2</b>

## 1 Introduction

The purpose of *widgetTools* is to provide a simple interface for users to build interactive widgets. Although the underlying implementation of any widget built using *widgetTools* is through the R *tcltk* package, users of *widgetTools* do not need to know all of the low-level **Tcl/Tk** commands, because the widget functionality is encapsulated in S4 classes which interact directly with standard R objects. For example, specifying the layout of some widgets (e.g. buttons and text labels) on a window can be as simple as supplying a list of lists, representing the list of rows of the grid of widgets, instead of using low level **Tcl/Tk** options to place and align each widget manually.

The design of *widgetTools* follows the Model-View-Controller pattern, described in [1]. This enables the separation of the information content of a widget (model) from the visual representation (view) and the actions/event-handlers associated with the widget (controller), in order to facilitate code reuse.

A basic primary widget class **basicPW** is implemented in order to create **pWidget** objects (instances of this class) which contain the application data (i.e. the *model*). A widget controller class **widget** is implemented to keep track of the various **pWidget** objects (in a nested list structure) and their associated actions/event-handlers. Finally, a **widgetView** class is implemented to store the low-level widget/window attributes, such as the ID attribute of an object created using the **tkoplevel** or **tkwidget** function in

the R *tcltk* package. These widget IDs are used to update the visual representation of the widget (view) if the internal data (model) changes.

The system is designed in such a way that users only have to deal with `pWidget` and `widget` objects. `widgetView` will be managed by `widget`.

## 2 Build a widget

The remainder of the vignette will be used to illustrate the construction of a dialog box which allows the user to:

- browse for files (using the `fileBrowser` function from the *tkWidgets* package),
- read some text labels displayed on the dialog,
- enter some text,
- select an option from a set of radio buttons,
- select an option from a listbox, and
- make some yes/no decisions using checkbuttons (checkboxes).

The basic steps are as follows:

1. Define an R environment for creating and manipulating `pWidget` objects, (rather than storing everything in the global environment).
2. Define several `pWidget` objects of different types (text label, text entry, button, listbox, textbox, radiobutton and checkbutton (checkbox)) using the functions `label`, `entry`, `button`, `listBox`, `textBox`, `radioButton` and `checkButton` respectively.
  - Each of these functions creates a `pWidget` object (of class `basicPW`) with appropriate slot values for the particular widget, e.g. the widget type slot (`wType`) would be “list” for a listbox.
3. Define `pWidgets`, a list of lists of `pWidget` objects in order to define the layout of widgets on the dialog, where the inner lists in the data structure correspond to rows of widgets on the dialog.
4. Use the `widget` function to construct and display a dialog containing the widgets in the `pWidgets` list
5. Use the `wValue` accessor function to extract the values specified by the user within the dialog and store them in standard R objects.

Shorter examples of using some of the *widgetTools* functions can be found in the help files for `makeViewer` and `tooltip`. The code shown in this vignette is a little longer, but illustrates the use of almost all of the most common widget elements in one dialog box.

Defining an R environment in which to manipulate `pWidget` objects has several advantages. Not only does it avoid the risk of overwriting/masking existing objects in the global workspace, but it has the advantage that objects from this environment can be obtained using the `get` function, thus avoiding the unnecessary copying of data in memory which would result from excessive passing (by value) of `pWidget` objects between functions every time they needed to be updated.

```
> library(widgetTools)
> PWEnv <- new.env(hash = TRUE, parent = parent.frame(1))
```

Then, we can create the `pWidget` objects that define the widget elements that are going to be rendered. A `pWidget` is an object of `basicPW` class defined below:

```
|-----|
|      basicPW      |
|-----|
| wName: string      |
| wType: string      |
| wValue: string     |
| wWidth: string     |
| wFuns: list        |
| wPreFun: function  |
| wPostFun: function |
| wView: widgetView  |
| wEnv: environment  |
| wNotify: list      |
|-----|
```

Where

- `wName` - a character string for the name to be associated with a given `pWidget`;
- `wType` - type of a Tk widget ("text" for text box, "list" for list box, "entry" for entry box, "button" for button, "radio" for radiobutton, "check" for checkbutton, "label" for text label,...);
- `wValue`: an undefined data type used to store information that will be displayed on the interface or updated using values obtained from the interface;

- `wWidth` - the physical width (in pixels) of the `pWidget` to be rendered. Applicable to most `pWidget` objects;
- `wFuns` - functions that will be associated with the given `pWidget` and invoked when a given operation to the `pWidget` takes place (e. g. clicked, get focused ...);
- `wPreFun` - the function defining the operations to be performed on the text of the `pWidget` before rendering the text;
- `wPostFun` - the function defining the operations to be performed on the text of the `pWidget` upon existing;
- `wView` - a `widgetView` object associated with each `pWidget` that will be used for updating the data displayed;
- `wEnv` - an R environment object where data updating and retrieval will take place;
- `wNotify` - a list of functions defining the actions to be performed when the value of the `pWidget` changes.

The following code creates several `pWidget` objects of different types:

```
> label1 <- label(wName = "label1", wValue = "File Name: ", wEnv = PWEnv)
> entry1 <- entryBox(wName = "entry1", wValue = "Feed me using browse",
+   wEnv = PWEnv)
> browse2Entry1 <- function() {
+   tempValue <- tclvalue(tkgetOpenFile())
+   temp <- get(wName(entry1), env = PWEnv)
+   wValue(temp) <- paste(tempValue, sep = "", collapse = ";")
+   assign(wName(entry1), temp, env = PWEnv)
+ }
> button1 <- button(wName = "button1", wValue = "Browse", wFuns = list(command = brow
+   wEnv = PWEnv)
> list1 <- listBox(wName = "list1", wValue = c(Option1 = TRUE,
+   Option2 = FALSE, Option3 = FALSE), wEnv = PWEnv)
> text1 <- textBox(wName = "text1", wValue = "Feed me something",
+   wEnv = PWEnv)
> label2 <- label(wName = "label2", wValue = "Select one: ", wEnv = PWEnv)
> radios1 <- radioButton(wName = "radios1", wValue = c(radio1 = TRUE,
+   radio2 = FALSE, radio3 = FALSE), wEnv = PWEnv)
> label3 <- label(wName = "label3", wValue = "Select one to many: ",
+   wEnv = PWEnv)
> checks1 <- checkButton(wName = "checks1", wValue = c(check1 = TRUE,
+   check22 = FALSE, check3 = FALSE), wEnv = PWEnv)
```

The `fileBrowser` function from the *tkWidgets* package is specified to be the action associated with pushing the **Browse** button (`button1`). The file name returned by the `fileBrowser` function is then displayed in the entry box (`entry1`) by assigning the filename to the `wName` slot in `entry1`. All the data manipulations will be done within the environment previously defined. At this time the environment does not contain the `pWidget` objects. They will be assigned to the environment by the system later. It is important that the `wName` for each widget element be the same as the name of the corresponding `pWidget` and to be unique. Values for `list1`, `radios1`, and `checks1` are named vectors. Names of the values will be displayed either as elements (for listboxes) or text descriptions (for radio and check buttons) of the corresponding widget elements to be rendered.

Event-handlers can be defined for each `pWidget` object, using the `wFuns` parameter of the appropriate `pWidget` constructor function, e.g. `button`. In the case of a button, the a function would be defined to be executed as the command for that button, and it would be specified in the `wFuns` list passed to the `button` constructor function as an attribute with name `command`. Any button defined is assumed to have a command element in its `wFuns` list. List boxes, radio buttons, check buttons, and text entries will have default functions defined for them to update their values when the corresponding widget element is pressed or text is changed. If you want anything additional to that, you will have to define a function and list that function in `funs`. Adding functions to the `wFuns` list is not fully implemented right now but will be available later.

Radio buttons and check buttons are defined as groups with the name and value of individual radio button or check button defined by the vector for value (see `radios1` and `checks1`).

`pWidget` objects have `set` and `get` functions to access and modify the variables. For example, we can get and set the value for the type of a `pWidget`:

```
> wName(label1)

[1] "label1"

> wName(label1) <- "YYY"
> wName(label1)

[1] "YYY"

> wName(label1) <- "label1"
```

To specify the layout of the widgets on the interface, the `pWidget` objects need to be put in a list of lists in such a way that the inner lists in the data structure correspond to rows in the grid of widgets on the interface. Each element (list) in the outer list will be treated as a unit and rendered one after another vertically and each element (`pWidget`) in the inner list will be treated as a unit and rendered one after another horizontally.

For example, if we want to have the first three `pWidgets` namely `label1`, `entry1`, and `button1` to appear in one row on top of the screen, `list1` and `text1` in one row below, `label2` and `radios1` in another row, and `label3` and `checks1` in yet another row, we will need to put the `pWidgets` in a list like the following. Naming of the lists for `pWidget` groups is arbitrary.

```
> pWidgets <- list(topRow = list(label1 = label1, entry1 = entry1,
+   button1 = button1), textRow = list(list1 = list1, text1 = text1),
+   radGroup = list(label2 = label2, radios1 = radios1), chkGroup = list(label3 = l
+   checks1 = checks1))
```

When we have the list of `pWidgets` ready, we can proceed to create a `widget` object that will construct a `widgetView` object to render the widget elements. A `widget` object is defined as follows:

```
|-----|
|      widget      |
|-----|
| pWidgets: list   |
| funs: list       |
| preFun: function |
| postFun: function|
| wTitle: string   |
| env: environment |
|-----|
```

where

- `pWidgets` - a list of lists with each element being a `pWidget` object to appear on the main interface represented by the `widget` object.;
- `funs` - a list of functions that will be associated with buttons on the interface to be created. The name of a function in the list will be the text which appears on the button and the function will be executed when the button is pressed;
- `preFun` - a function that will be executed when the Tk widget is constructed;
- `postFun` - a function that will be executed when the Tk widget is destroyed.
- `wTitle` - a character string for the title to be displayed in the title bar of the main interface represented by the `widget` object.

- `env` - an R environment object within which the original values for the main `widget` object will be stored and updating and retrieval of the values will take place

The following code constructs a `widget` object. In practice it is not necessary to use `if(interactive()){...}`. This is only included so that this vignette can be constructed automatically using `Sweave` without necessarily having a graphics device open.

```
> if (interactive()) {  
+   aWidget <- widget(wTitle = "A test widget", pWidgets, funs = list(),  
+     preFun = function() print("Hello"), postFun = function() print("Bye"),  
+     env = PWEnv)  
+ }
```



Now, we can change the values of the `pWidget` objects we have created through the interface. Click the **Browse** button to select a file using the `fileBrowser` widget from the `tkWidgets` package. After clicking the **Finish** button on the `fileBrowser` widget (not shown), the File Name entry box on the test widget above will contain the file name which was selected (and returned by the `fileBrowser` function). Try selecting an item from the list box, selecting a radio button, toggling the check buttons, and typing in a few words in the text box. Then click the **Finish** button on the test widget, and a `widget` object will be constructed. The `widget` object contains `pWidget` objects, the value of whose elements will be modified depending what has been performed on the interface we just played with. The values for `entry1`, `list1`, `text1`, `radios1`, and



checks1 now become:

```
> if (interactive()) {  
+   wValue(pWidgets(aWidget)[["topRow"]][["entry1"]])  
+   wValue(pWidgets(aWidget)[["textRow"]][["list1"]])  
+   wValue(pWidgets(aWidget)[["textRow"]][["text1"]])  
+   wValue(pWidgets(aWidget)[["radGroup"]][["radios1"]])  
+   wValue(pWidgets(aWidget)[["chkGroup"]][["checks1"]])  
+ }
```

## References

1. MVC Krasner, G. and Pope, S. 1988. Huynen, M. A. and Bork, P. 1998. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.