The traitr package

John Verzani
verzani at math dot csi dot cuny dot edu

February 2, 2010

Abstract

This vignette describes the `traitr` package. This package provides an interface to GUI design within R in the spirit of the `Traits UI` python module. The basic idea is that the user specifies the type of data that is desired, and the package provides a reasonable GUI to edit that data.

1 Five-minute, or so, introduction

The `traitr` package is provided to provide a natural, relatively easy means to produce GUIs for functions. The design was inspired by the `Traits UI` python modele (`http://code.enthought.com/projects/traits`), although the implementation is not nearly as feature rich as that.

The `Traits UI` interface uses the model-view-controller design pattern. The main idea is that a user specifies a model consisting of possibly many properties, each holding data of a certain type. Each property we call an item. A view of the model is specified by indicating which items are to be shown along with a layout. The items themselves know how to present themselves graphically so that the user can interact, or edit, the values. One then ties the model and view together by specifying some common methods (a controller).

This provides a simple map between a function with arguments (whose values are of a certain type) and a GUI to edit these values, which makes the writing basic GUI dialogs for R functions very straightforward – no GUI programming is needed. [1]

The `traitr` package uses `gWidgets` to provide a link to a graphical toolkit. We've only used `RGtk2` in developing this package, `tcltk` was slower and doesn't have the graphics device. This package must be installed separately and may involve installing a toolkit.

There are a few means available now to have GUIs automatically written for you such as `fgui` and the `ggenericwidget` function in `gwidgets`. The basic idea employed is an inspection of a function's arguments to determine their type and provide the appropriate widget for that type. This package does not attempt to do this, rather the user specifies the type for the arguments. The basic unit in `traitr` is an item, which is an object that knows what type of value it should hold and how to represent that value in a GUI. Since the programmer knows the type of value they want, this isn't difficult to provide, yet the GUI creation aspect is mostly hidden from sight.

[1] This vignette documents version 0.4 of the package. At this point, the interface is experimental and subject to change.

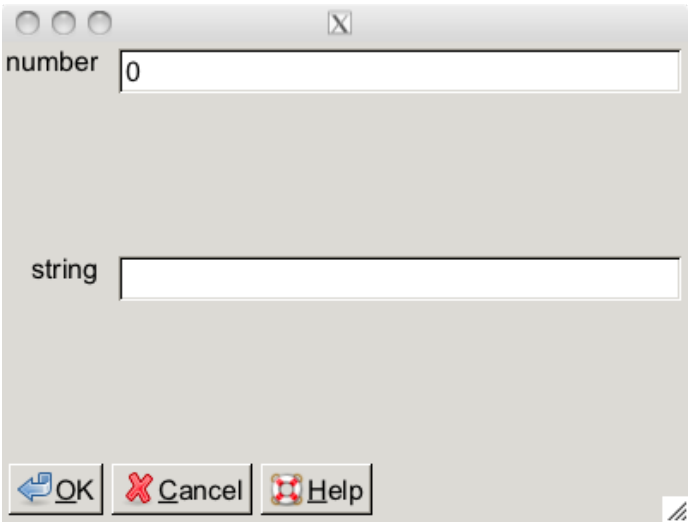

Figure 1: Basic interface with two items, a number and string.

At this point, there are a few basic type of items listed in Table 2. Adding more is on the TODO list.

To begin with traitr, we load the package and specify a gWidgets implementation.

```
require(traitr)
require(gWidgets)
options(guiToolkit="RGtk2")
```

An item constructor is called with a default value and possibly many other values. To see how a simple GUI can be made, here we construct a GUI to collect a number and string:

```
dlg <- aDialog(items=list(
                 number=numericItem(0),
                 string=stringItem("")
                 )
               )
dlg$make_gui()                          # method call using $ notation.
```

The aDialog constructor returns a proto object for which there are many methods defined, but just a few that are necessary to know. In this instance, we use the dialog's make_gui method to create the GUI. To see other documented methods and properties of the proto objects produced, call their show_help method. [2] That is, this command will produce a web page with the information for the dlg instance: dlg$show_help().

[2]The proto package extends R's environments to implement a prototype programming style. To call a function, we use the $ notation to reference the function. Unlike storing a function in a regular environment, this function call passes back into the function, as the first argument, a reference to the proto object. As such, the function signature would be something like (self, arg1, arg2), where self is used here for familiarity with javascript (another prototype language). However, it is traditional to use . as this variable instead of self, so we use this in the code, as needed.

The default GUI has three buttons, an OK button (which when clicked prints out the values in the GUI), a Cancel button to dismiss the dialog and a Help button to call up a simple help string, which in this example is not implemented. To change the buttons, one assigns the names to the `buttons` property of the dialog. The values "Undo" and "Redo" will implement the undo/redo pattern for the underlying model.

To override the default behaviours or to assign actions to other buttons, one defines methods for the dialog object using a certain naming convention. For the dialog buttons, if one defines a function (`buttonname_handler`), then this will be called when the button "buttonname" is clicked. For example, to make the dialog do something else, we have:

```
dlg <- aDialog(items=list(
                  number=numericItem(0),
                  string=stringItem("")
                  ),
                OK_handler=function(.) { # . is reference to dlg object
                  values <- .$to_R()
                  f <- function(number,string)
                    cat("number is",number, "string is",string,"\n")
                  do.call(f, values)
                }
                )
dlg$make_gui()
```

The function `OK_handler` becomes a method of the `proto` object, so its lone argument `.` passes into the function body the `proto` object. (These objects are environments, so the usual pass by copy is not used, rather this is the same object.) Its method `to_R` returns a named list of the values stored for each item in the dialog – which is convenient for `do.call` – and this handler simply echoes them back in a friendly manner.

To make a less trivial dialog we just need to have more complicated functions. What follows is a dialog (Figure 2) for computing a t-test when the values are summarized. First a function to compute the p-value.

```
basic.t.test <- function(mean, mu, sd, alternative=c("two.sided","less","greater"),
                         n, ...) {
  alternative <- match.arg(alternative)
  obs <- (mean - mu)/(sd/sqrt(n))
  switch(alternative,
         "greater" = 1 - pt(obs, df=n-1),
         "less" = pt(obs, df=n-1),
         2*(1 - pt(abs(obs), df=n-1))
         )
}
```

Our dialog setup is similar, but we have more variables that need their type specified.

```
dlg <- aDialog(items=list(
                  mean = numericItem(0),
```

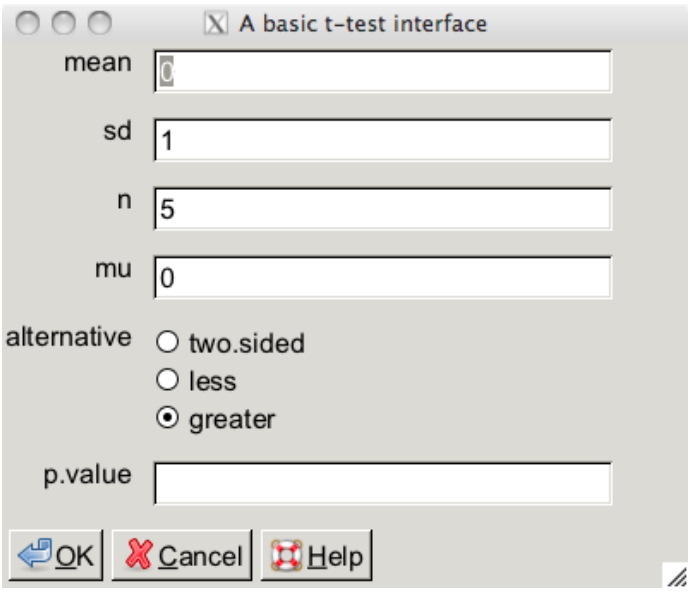

Figure 2: Basic t-test interface

```
    sd = numericItem(1),
    n = numericItem(5),
    mu = numericItem(0),
    alternative=choiceItem("two.sided", values=c("two.sided","less","greater")),
    output = stringItem("", label="p.value")
    ),
  title="A basic t-test interface",
  help_string="Enter in summarized values then click OK to get the p-value",
  OK_handler=function(.) {
    lst <- .$to_R()
    lst$output <- NULL     # not really needed, ignored by function
    out <- do.call("basic.t.test", lst)
    .$set_output(out)
  }
  )
dlg$make_gui()
```

Here we used the `choiceItem` to provide a choice to the user. As well, we added values for the title and help string. The handler has the method call `set_output` to set the value to show in the output widget. When a dialog is created, both "`get`" and "`set`" methods are created for each item specified, where `output` matches the `name` property of the item (which is specified through the named list passed to `items`).

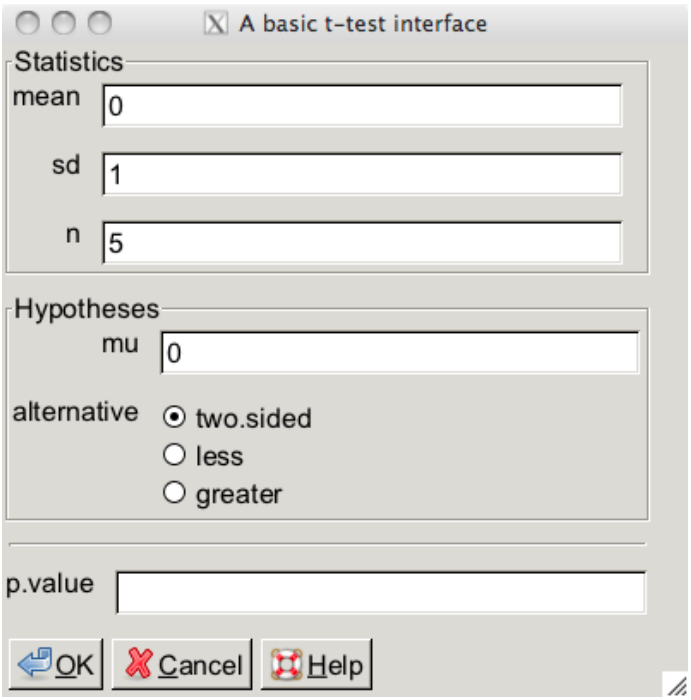

Figure 3: The *t*-test interface using an alternate view that separates off the statistics from the hypotheses.

1.1 Views

The default GUI layout uses a 2-column table to organize the items. To modify this, one creates an alternate view or layout for the items. There are a few view constructors, similar to the containers in `gWidgets`. Table 5.1 lists them.

The basic layout is equivalent to:

```
view <- aContainer("mean","sd","n","mu","alternative","output")
```

The `aContainer` constructor simply lays out its child components using a two-column table, one for the label and one for the widget representing the item. The items are specified by name. (The name can be explicitly set when the item is constructed, e.g. `numericItem(1, name="x")` or, as in the example, found from using a named list to define the items.)

To make a more complicated layout, we nest the containers. Here is how one can separate out the computed values from the hypotheses using frames:

```
view <- aContainer(aFrame("mean","sd","n", label="Statistics"),
                   aFrame("mu", "alternative", label="Hypotheses"),
                   "output")
```

However, this view will not align values, as the `aFrame` container is just a box container. Instead we add in another level with `aContainer` (Figure 3):

```
view <- aContainer(aFrame(aContainer("mean","sd","n"), label="Statistics"),
                   aFrame(aContainer("mu","alternative"),label="Hypotheses"),
                   separatorItem(),
                   "output")
```

To see this in action, to save typing, we use the `instance` method, which for this purpose simply copies the dialog object. Then to specify a view different than the default, we use the `gui_layout` argument of the `make_gui` method:

```
dlg1 <- dlg$instance() ## instead of copying the definition above.
dlg1$make_gui(gui_layout=view)
```

1.2 Validation

Items use the model-view-controller design pattern, with a transport phase from view to model that allows for inspection of values. As such, we can check for valid input when assigning from the view to the model. [3] When an item has invalid input, some signal is specified. To see it, try typing in a non-numeric value in where a number is expected.

To see how to specify our own validation function, [4] we first write the function, then add it to the item. For the standard deviation, we shouldn't be able to specify non-positive standard deviations. A function to check this would be:

```
positive_value <- function(., rawvalue) {
  value <- as.numeric(rawvalue)
  if(!value > 0)
    stop("value is not positive")
  value
}
```

The `.` argument is the item instance (not the dialog, say) and the argument `rawvalue` comes from the widget. (In this case a string, so we coerce the value before testing.) The validation functions should throw an error with a message if not valid. (The item itself has a `coerce_with` method to carry out the proper coercion.)

To add to a numeric item, we assign to its `validate` method: [5]

```
sd <- numericItem(1, name="sd")
sd$validate <- positive_value
```

To do this for our dialog object, we can grab the item from the dialog, then specify its validation method:

```
sd <- dlg$get_item_by_name("sd")         # lookup and return item by name
sd$validate <- positive_value            # assigns method to item
```

[3] Although we allow invalid input for technical reasons. It isn't until the call to `to_R` that an issue will arise.

[4] Validation is done for the main value stored in an item. For more complicated items using more than one main property, one can defind `validate_PROPERTYNAME` methods.

[5] Here we see the ease of using the `proto` package, but it is also a burden. It is quite possible to assign to a "private" method, thereby having very unintended side effects that can be hard to debug.

```
dlg1 <- dlg$instance()
dlg1$make_gui(gui_layout=view)
```

1.3 Conditional layout

Parts of a view can be enabled conditioned on other parts of the GUI. In this example we make a GUI (Figure 4) for a one or two-sample t-test. As some options (such as `paired`) only make sense when two samples are specified, we enable those only when a possible value for `y` is specified. In the `x` and `y` items, the argument `eval=TRUE` is specified, which causes the specified value to be parsed and evaluated in the global environment. This allows the user to specify a variable name or R expression for the variable, and not just a number.

```
dlg <- aDialog(items=list(
                  x = numericItem(NA, eval=TRUE),
                  y = numericItem(NA, eval=TRUE),
                  alternative=choiceItem("two.sided",
                    values=c("two.sided", "less", "greater")),
                  mu = numericItem(0),
                  paired=trueFalseItem(FALSE),
                  var.equal=trueFalseItem(FALSE)
                  ),
                title="GUI with some parts disabled"
                )
view <- aContainer("x",
                   aContext("y", context=dlg,
                            enabled_when=function(.) { # y depends on x
                              ## . here is the context value, not the container object
                              val <- .$to_R()$x
                              !is.null(val) && !is.na(val)  && (nchar(val) > 0)
                            }),
                   "alternative",
                   "mu",
                   aContainer("paired","var.equal", context=dlg,
                              enabled_when=function(.) {
                                val <- .$to_R()$y
                                !is.null(val) && !is.na(val) &&  (nchar(.$get_y()) > 0)
                              })
                   )
dlg$make_gui(gui_layout=view)
```

The `aContext` container is used to specify optional arguments to the view, but does not do any layout. In this case, the `y` value is conditioned on the value specified for `x`. The `context` argument specifies a dialog (or item group) where the variable is to be looked up.

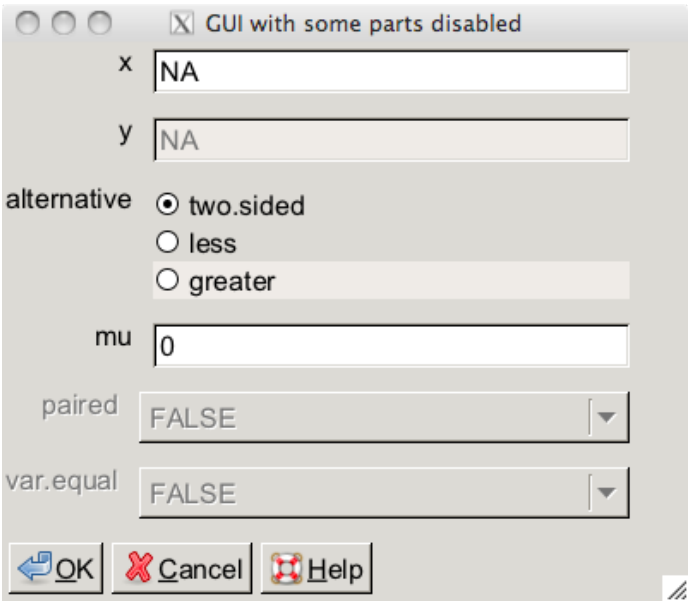

Figure 4: A t-test interface with some items having sensitivity depending on values of other items.

1.4 Why wait?

The basic dialog calls a function when the OK button is clicked. Making a more interactive GUI, say to update when a value is changed, can be done by creating a method `model_value_changed`. For example, here is a GUI (Figure 5) to draw a graph for a random sample of size n.

```
drawGraph <- function(n,..) hist(rnorm(n))
dlg <- aDialog(items=list(
                  n=rangeItem(10, from=1, to=100, by=1),
                  graph=graphicDeviceItem()
                  ),
                title="Draw a graph",
                help_string="Adjust slider or click OK to produce a new graph",
                model_value_changed=function(.) {
                  l <- .$to_R()
                  l$graph <- NULL        # not really necessary
                  do.call("drawGraph", l)
                },
                OK_handler=function(.) do.call("drawGraph",.$to_R())
                )
dlg$make_gui()
```

The dialog is a model that observes itself for changes. When such a change occurs, methods matching a certain naming convention are called. The `model_value_changed` is always called.

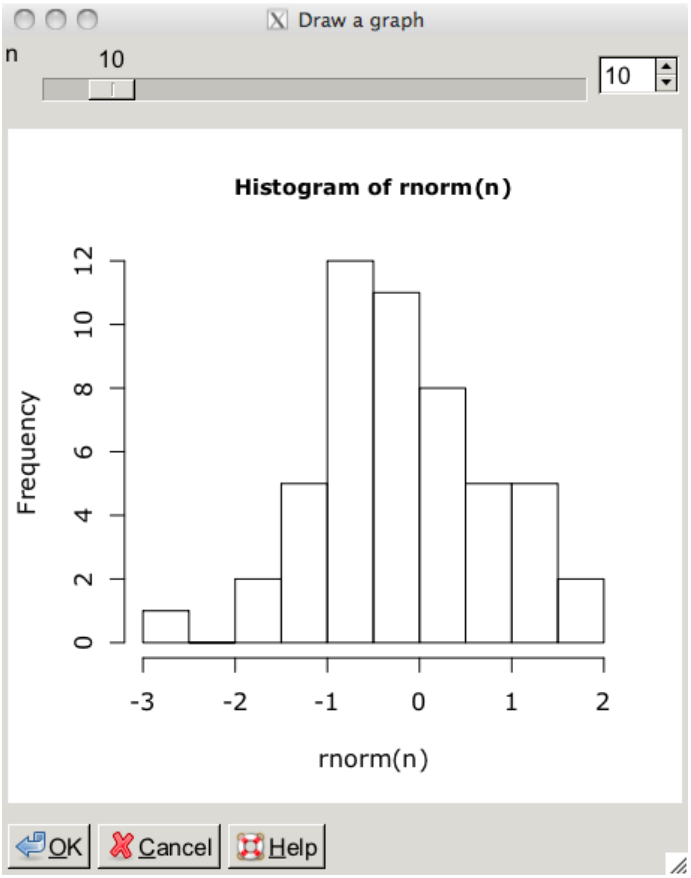

Figure 5: GUI where slider movement updates the graphic.

Table 1: Table of item constructors.

Constructor	Description
stringItem	For holding strings
numericItem	For numbers
integerItem	For integers
expressionItem	For R expressions
trueFalseItem	For Boolean values
choiceItem	For choosing one or more values from a list of possible values
rangeItem	To select a value from a range of values
buttonItem	For adding a button
labelItem	For adding a label
dateItem	For editing a calendar date.
separatorItem	To add a visual separator
dataframeItem	To select a data frame
variableSelectorItem	To select a variable from a data frame
graphicDeviceItem	(RGtk2 only) To embed a graphic device
formulaItem	For formula specification (to be written)
dfeditItem	To edit a data set (to be written)

2 Items

The basic object in `traitr` is an item. An item includes a model to keep track of its value(s), an editor (the view) to create a GUI to edit the value(s) and a controller to link the two. In addition, items may include methods for validation of its values. Following the design of `Traits UI`, items allow the programmer to focus on the type of value, not the GUI presentation of the value.

Items are constructed through one of the item constructors. Table 2 lists the currently available ones. The constructors have some common arguments: `value` to specify the initial value of the item; `name` to specify the unique name for the item within a dialog (this can also be specified through a named list); `label` for a text label to identify the argument (defaults to `name`); `tooltip` to specify a tooltip for the widget, should it provide one; `attr` to pass arguments to the `gWidgets` constructor underlying the editor; `model` to specify a model for the item (models can be shared, or a new one will be created); and `editor` to specify a different editor.

In addition to these, each item has some item-specific arguments. For example, `numericItem` and `integerItem` have an argument `eval_first` which if set to `TRUE` will force the value from the editor to be evaluated within the global environment as a string.

The `choiceItem` provides a means to choice a value for a specified list of values. The editor may use a radio button group, a checkbox group, a combobox or a table widget to present the choice depending on the size of the list of values and whether multiple selection is possible. The constructor has two arguments, `value` to specify the initially chosen value and `values` to specify the list to choose from. The former can be specified by value (the default) or by index (if `by_index=FALSE` is give). The latter may be a vector or data frame.

The `dataframeItem` constructor provides a means to select a data frame. The `variableSelec-`

torItem constructor requires such an object to be passed to it, in order for it to have a context to find names in. For example

```
dfi <- dataframeItem(value=".GlobalEnv", name="dfi",
                     editor_style="compact") # alternative editor style
dlg <- aDialog(items=list(
                  dfi,
                  vsi=variableSelectorItem("", multiple=FALSE, dataframeItem=dfi,
                    attr=list(size=c(200,200)))
                  ))
dlg$make_gui()
```

2.1 Item groups

An ItemGroup is used by aDialog to combine more than one item into a unit which acts as the model for the GUI. The anItemGroup constructor is similar to that for a dialog, but is used if one wants to embed the resulting GUI into some gWidgets container without the buttons, menubar, and toolbar options.

Item groups implement the model interface, described in the next section.

Item groups can allow for resuse of parts of a GUI. In this example, we see how we can separate some pieces out that are common to both a *t*-test and a Wilcox test call:

```
hyps <- anItemGroup(items=list(
                      mu=numericItem(0),
                      alternative=choiceItem("two.sided", c("two.sided","less","greater"))
                      ),
                    gui_layout=aFrame("mu","alternative", label="Hypotheses")
                    )
ttestDialog <- aDialog(items=list(
                         x=numericItem(NA, eval=TRUE),
                         y=numericItem(NA, eval=TRUE),
                         hyps$instance()
                         ),
                       OK_handler=function(.) {
                         do.call("t.test",.$to_R())
                       }
                       )
wilcoxDialog <- aDialog(items=list(
                          x=numericItem(NA, eval=TRUE),
                          y=numericItem(NA, eval=TRUE),
                          hyps$instance()
                          ),
                        OK_handler=function(.) {
                          do.call("wilcox.test", .$to_R())
                        }
```

```
                        )
ttestDialog$make_gui()
wilcoxDialog$make_gui()                 # shares alt info!
```

The use of `instance` ties the models together, but allows for multiple views. This may not be desirable. If two independent realizations are desired, then creating a factory (a function) to produce new instances of `hyps` would be suggested.

3 Models

Each item (and item group) implements a model interface. The underlying items uses the model-view-controller design pattern. A model is a object with a means to store and access properties and a mechanism to notify observers when these properties have changed.

3.1 Getters/Setters

Each model property has `get` and `set` methods to interact with the value stored in the model. For an item with a given name, say `x`, there are always methods `get_x` to get the value and `set_x` to set the value. The get method returns a raw value, unlike the method `to_R` which returns a value after coercion to the appropriate type. For example:

```
i <- numericItem(0, name="item1")
i$get_item1()
```

```
[1] 0
```

```
i$set_item1(3)
i$get_item1()
```

```
[1] 3
```

```
try(i$set_item1("c(1,2,3)"))            # fails validation, still stored in model
```

```
Not a valid value: Error in function (., value)  :
  (converted from warning) NAs introduced by coercion
```

```
i$get_item1()
```

```
[1] "c(1,2,3)"
```

Where as, we can have the value run through `eval` to have:

```
i <- numericItem(0, name="item2", eval=TRUE)
i$set_item2("c(1,2,3)")                 # now okay
i$get_item2()
```

```
[1] "c(1,2,3)"
```

```
 i$to_R()                                  # coerced
```

```
$item2
[1] 1 2 3
```

Item groups and dialogs also implement these methods, as they act as models with several properties:

```
 dlg <- aDialog(items=list(
                  x=numericItem(0),
                  y=stringItem("a")
                  ))
 dlg$get_x()
```

```
[1] 0
```

```
 dlg$set_y("some string")
 dlg$get_y()
```

```
[1] "some string"
```

3.1.1 Getting/Setting other values in an item

If an item in an item group has more than one main property, the main getter/setter pairs created from each item name do not suffice. For example, the choice item has a main property for setting the value, but another property to store the values to choose from. To access a secondary property, we first extract the item by name with the `get_item_by_name` method of item groups, and then use the item's getter/setter values. For example:

```
 ig <- anItemGroup(items=list(
                   x=numericItem(1),
                   y=choiceItem("a", values=letters[1:5])
                     )
                   )
 ig$get_y()
```

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

```
 i <- ig$get_item_by_name("y")
 i$get_y()                                 # same as above
```

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

```
 i$get_values()                            # get values
```

```
[1] "a" "b" "c" "d" "e"
```

```
 i$set_values(letters)                     # to set values
```

3.2 Observers

Models implement the observer pattern. An observer is notified whenever a model property changes and if the model changes. To do so, the observer (which is a model instance like an item, item group, or dialog; or a controller) would have methods `model_value_changed` which is called whenever any property is changed or `property_NAME_value_changed` which is called when a given property is changed.

To add an observer to a model, the `add_observer` method is used, whereas `remove_observer` is provided to remove the observer. This example defines two dialogs, the second with a suitably named method and then has the second observe changes in the first.

```
dlg <- aDialog(items=list(
                 x=numericItem(0),
                 y=numericItem(0)
                 )
               )
dlg1 <- aDialog(items=list(
                  a=numericItem(0)
                  ),
                property_x_value_changed=function(., value, old_value) {
                  .$set_a(.$get_a() + value) # add value to a (assumes numeric)
                }
                )
dlg$add_observer(dlg1)
dlg1$get_a()
```

```
[1] 0
```

```
dlg$set_x(10)
dlg1$get_a()                            # updated by x
```

```
[1] 10
```

A dialog listens to itself, which allows the change of one property to update another. This example shows how the image is updated whenever the file is changed.

```
dlg <- aDialog(items=list(
                 f=fileItem(""),
                 i=imageItem("",attr=list(size=c(480,480)))
                 ),
               property_f_value_changed=function(., value, old_value) {
                 .$set_i(value)
               },
               buttons="Cancel")
dlg$make_gui()
```

The `model_value_changed` method has previously been illustrated. This method is called whenever any value is changed in the model.

3.3 Sharing a model

An item can share a model with another item. This allows for synchronization of values in a GUI. There are several means to do this: specifying the model at construction, calling the `set_model` method, or calling the `instance` method. For example,

```
i <- numericItem(0, name="x")
j <- numericItem(1, name="x")
j$get_x()
```

```
[1] 1
```

```
j$set_model(i)
j$get_x()
```

```
[1] 0
```

```
i$set_x(10)
j$get_x()
```

```
[1] 10
```

The `instance` method also creates a new item with a shared model, or as seen, a new dialog with shared models. There can only be one GUI for an instance of an item, but different instances can have GUIs, allowing multiple views for the same model.

4 Dialogs

Dialogs and item groups combine several items into a model, with different properties. In addition to the getter/setter methods, there is the useful `get_item_by_name` to retrieve a given item, and `to_R` to get the values as a named list.

A dialog is an item group which creates its own top-level window. As such, dialogs have options for decorating the window: e.g., to add menubars, toolbars, and statusbars etc. We've seen how specifying the title and help string at construction is done through the argument `title` and `help_string`.

The top-level container is constructed by a call to the `make_gui` method. To close the GUI programmatically, the `close_gui` method is available.

4.1 Menubars, toolbars, statusbars

Menu bars and toolbars are specified using `gWidgets` objects. Basically, a list of `gaction` objects (of `gWidgets`) are specified to the dialog properties `menu_list` and `toolbar_list`.

```
mb_l <- list(File=list(
              New=gaction("new", icon="new", handler=function(h,...) print("New")),
              Quit=gaction("quit", icon="quit", handler=function(h,...) dlg$close_gui())
```

```
                ))
tb_l <- list(Quit=gaction("quit", icon="quit", handler=function(h,...) dlg$close_gui()))
dlg <- aDialog(items=list(x=stringItem("some value")),
               menu_list=mb_l,
               toolbar_list=tb_l,
               title="Dialog with menu and toolbar")
dlg$make_gui()
```

A status bar is created if the value of the `status_text` property is non `NULL`, say `""`, when `make_gui` is called. If there is a status bar, the method `update_status_text` is provided.

4.2 Buttons

Dialogs have a default set of buttons: OK, Cancel and Help. When clicked these call the methods `OK_handler`, `Cancel_handler` and `Help_Handler`. As well, buttons named Undo and Redo have default handlers if specified. Likely, only `OK_handler` will need to be overridden. An example was previously given.

To specify other buttons, simply provide the names to the `buttons` property prior to construction of the GUI. For example, to have a GUI whose closure depends on the value of x being changed, we might have:

```
dlg <- aDialog(items=list(
                 x=numericItem(0)
                 ),
               title="Change x to be able to close",
               buttons="Close",
               Close_handler=function(.) {
                 if(.$get_x() != 0)
                   .$close_gui()
                 })
dlg$make_gui()
```

If a button name has spaces or other non-alpha characters, they are stripped before calling the handler. For example, a button labeled ”click me” would call the handler `clickme_handler`.

The special button name “SPACE” will insert a 10 pixel space. The special button name “SPRING” will insert a “spring” between the buttons.

5 Views

View provide a means to customize the look of a dialog or item group. A view is simply a container. The constructors specify the child components as either strings indicating items, or other views. A `context` allows the appropriate item to be found from the string. By default, this context is the calling dialog or item group, but need not be. This can be useful if the item to display is in another item group.

Table 2: Table of view constructors.

Constructor	Description
aContainer	Basic container, uses tabular layout
aTableLayout	Tabular layout with more than 2 columns
aGroup	Box container to pack in children left to right or top to bottom
aFrame	Box container with decorative frame
anExpandGroup	Box container with trigger to hide
aPanedGroup	Two pane container
aNotebook	Notebook container
aContext	Provide context for an item or items

5.1 Types of views

The various view are described by their man pages. They are very similar to the `gWidgets` containers except for that provided by the `aContext` constructor. This view does not place its items in a container, but rather provides a means to specify a context. This is useful if an item comes from another item group, but you wish the alignment of the item to be as others.

5.2 Enabled and visible

In a previous example we illustrated the view method `enabled_when` which is called whenever a property value is changed. If this returns `FALSE`, then this part of the view has its sensitivity disabled.

Similarly, there is a `visible_when` method that can be used to hide a portion of a GUI until something occurs.

5.3 Editors

A view is used to layout one or more items, whereas each item itself has its interface produced by an editor. Editors are specified by the item constructor, with a default editor usually being appropriate. Some items provide more than one possible editor, for instance the `choiceItem`. To specify which editor to use, the property `editor_style` is set to the name of the different editor.

To define a new item, one must also provide an editor or use one of the existing ones.

5.4 Integration with `gWidgets`

The `gWidgets` package is used to provide the link between the `traitr` objects and a graphical toolkit package, such as `RGtk2`. In order to customize the appearance of the editors, one passes through the `attr` argument a list of values for `gWidgets` constructors.

To place a `traitr` GUI within a `gWidgets` GUI, one can use either the `make_ui` method of an item obect, or the `make_gui` method of an itemgroup object, as both have a `container` argument to pass in a `gWidgets` container.

To place a gWidgets object within a traitr object is more difficult, as there is no API for this (yet). At this point, one can create a container object and place it within that. The container object has a property container that stores the gWidgets container needed. However, this isn't available until after the GUI has been drawn.

This idea is illustrated below:

```
dlg <- aDialog(items=list(x=numericItem(1)))
g <- aGroup()                          # define outside view to access later
view <- aContainer("x",g)
dlg$make_gui(gui_layout=view, visible=FALSE) # postpone showing, but create containers
l <- glabel("Look ma, a gWidgets label", cont = g$container) # how to find container
dlg$visible(TRUE)
```

5.5 Adding elements to an already drawn GUI

In gWidgets, and indeed only because it is a common feature of the graphical toolkits, one can add elements to a previously drawn GUI. In gWidgets, this is done through the add method of containers (although this is usually called indirectly in the containers construction).

However, to add to a GUI using traitr is not so direct. Using the previous idea though allows it to be done provided you provide a place where objects will be added in the GUI construction. This is implemented by hiding the visibility of the container through its visible_when method.

```
dlg <- aDialog(items=list(x=numericItem(0)))
g <- aGroup(visible_when=function(.) FALSE) # suppress showing
view <- aContainer("x", g)
dlg$make_gui(gui_layout=view)
## now to add to the GUI at g:
ig <- anItemGroup(list(y=stringItem("a string")))
ig$make_gui(container=g)
g$visible_when <- function(.) TRUE
dlg$update_ui()
```

The call to the update_ui method is also done whenever a value changes, but in this case we want it to be done immediately so the string item shows up.

6 Some examples

Some additional examples follow that are a bit more involved.

6.1 chooseCRANmirror

The function chooseCRANmirror provides a GUI for selecting a CRAN mirror, providing a table to choose the mirror from and a button to click once the selction is done. Here we give a slightly better alternative, with a bit more information and also connect to a double click event. The key function, setCran, is taken from the original with slight modification.

```
m <- getCRANmirrors(all = FALSE, local.only = FALSE)[,c(1,2,4)]
setCran <- function(.,...) {
  URL <- .$get_cran()
  repos <- getOption("repos")
  repos["CRAN"] <- gsub("/$", "", URL[1L])
  options(repos = repos)
  .$close_gui()
}
```

Now we define our model, and use the default view.

```
dlg <- aDialog(items=list(
                 cran=choiceItem(value=NA, values=m,
                   show_label=FALSE,  # suppress label
                     attr=list(chosencol=3, size=c(400,500)) #chosencol is URL one, not first
                   )
                 ),
               title="Choose a CRAN Mirror",
               help_string="Click a mirror, then OK, or double click on mirror",
               OK_handler=setCran,                 # OK button click
               property_cran_value_changed=setCran # double click
               )
dlg$make_gui()
```

6.2 The tkdensity dialog

The `tkdensity` demo is a standard in the `tcltk` package. Here we redo the demo using `traitr` (Figure 6). This function will make the density plot for us, in a manner similar to that of the demo.

```
  replot <- function(.) {
    l <- .$to_R()
    f <- function(dist, kernel, n, bw,...) {
      y <- switch(dist, "Normal"=rnorm(n), rexp(n))
      plot(density(y,  bw=bw, kernel=kernel), xlim=range(y)+c(-2,2), main="Density example")
      points(y, rep(0,n))
    }
    do.call(f,l)
  }
```

Here we define the items and the dialog properties. There are 5 main items, including the graphic device. The labels are suppressed, as we use a frame to set off similar to the original example.

```
dlg <- aDialog(items=list(
                 dist=choiceItem("Normal", values=c("Normal","Exponential"),
                   show_label=FALSE),
                 kernel=choiceItem("gaussian",
```

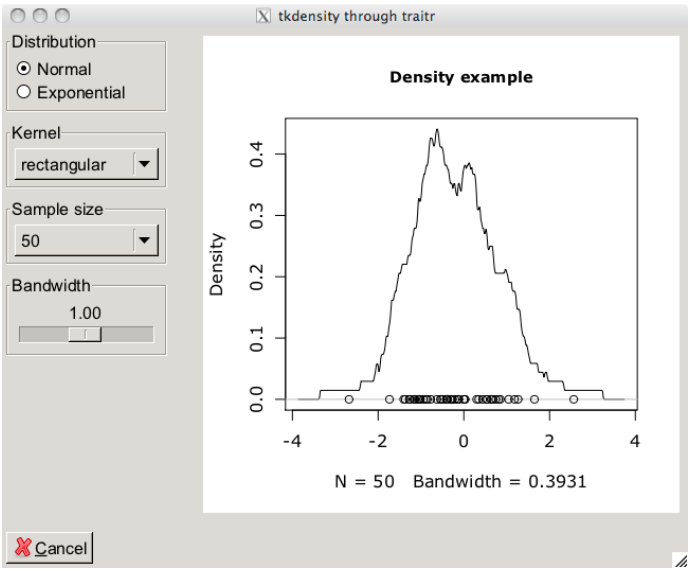

Figure 6: A dialog similar to that of the tkdensity demo in the tcltk package.

```
    values=c("gaussian", "epanechnikov", "rectangular",
      "triangular", "cosine"),
    show_label=FALSE),
  n=choiceItem(50L, as.integer(c(50,100,200,300)),
    show_label=FALSE),
  bw=rangeItem(value=1, from=0.05, to=2.00, by=0.05,
    show_label=FALSE),
  out=graphicDeviceItem()
  ),
help_string="Adjust a parameter to update graphic",
title="tkdensity through traitr",
buttons="Cancel",
model_value_changed=replot)
```

Our layout places the controls on the left within frames, and the graphic device on the right. We use only the 'Cancel' button to match, somewhat, the demo.

```
view <- aGroup(aContainer(aFrame("dist", label="Distribution"),
                          aFrame("kernel", label="Kernel"),
                          aFrame("n", label="Sample size"),
                          aFrame("bw", label="Bandwidth")
                          ),
               "out",
               horizontal=TRUE)
dlg$make_gui(gui_layout=view)
```

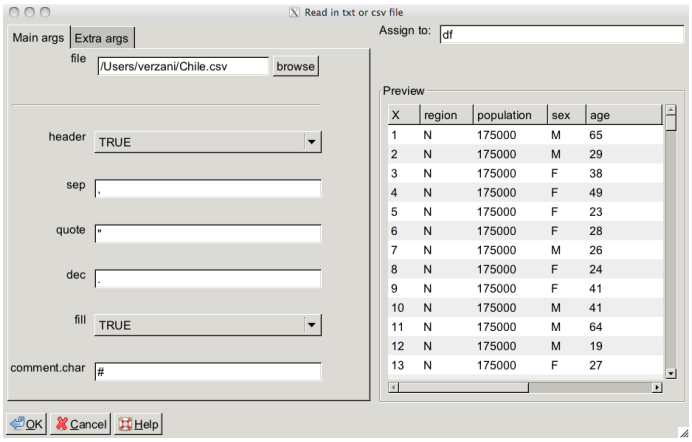

Figure 7: A GUI for the `read.csv` and `read.table` functions.

```
replot(dlg)                          # initial plot
```

6.3 A GUI for read.csv and read.table

This example shows how the mapping between argument and `traitr` items proceeds in a straightforward manner. We provide a GUI (Figure 7) for `read.csv` and `read.table`. While these both work quite easily with the default arguments, in the case they don't the arguments can be numerous. This GUI allows one to specify the arguments (except for typing in names and class types, although that could have been implemented using `stringItem`s which evaluate their argument first.

First, the model part:

```
dlg <- aDialog(items=list(
                file=fileItem("", attr=list(
                                  filter=list("CSV or TXT"=list(
                                                patterns=c("*.csv","*.txt")
                                                ),
                                    "All files" = list(patterns=c("*"))
                                    ))),
                header=trueFalseItem(TRUE, tooltip=paste("Variable onfirst line?")),
                sep=stringItem("", tooltip="Field separator character"),
                quote=stringItem("", tooltip="Set of quoting characters"),
                dec=stringItem(".", tooltip="Character used for decimal points"),
                as.is=trueFalseItem(!default.stringsAsFactors(),
                  tooltip="Do not convert character to factor"),
                na.strings=stringItem("NA", tooltip="Strings to be NA", eval=TRUE),
                nrows=integerItem(-1, tooltip="Max number of rows to read"),
                skip=integerItem(0, tooltip="Number of lines to skip at header"),
                check.names=trueFalseItem(TRUE, tooltip="If TRUE ensure names are valid"),
```

```
    fill=trueFalseItem(TRUE, tooltip="Fill unequal length rows if TRUE"),
    strip.white=trueFalseItem(TRUE),
    blank.lines.skip=trueFalseItem(TRUE, tooltip="If TRUE, skip blank lines"),
    comment.char=stringItem("#", tooltip="Comment character"),
    allowEscapes=trueFalseItem(TRUE, tooltip="C-style escapes read verbatim?"),
    stringsAsFactors=trueFalseItem(default.stringsAsFactors(),
      tooltip="Characters converted to factors"),
    fileEncoding=stringItem(""),
    encoding=stringItem("unknown"),
    ## our things
    assign.to=stringItem("df", label="Assign to:"),
    output=tableItem(attr=list(size=c(400,400)), show_label=FALSE),
    file.type=stringItem("")
    ),
  title="Read in txt or csv file",
  help_string="Select a file, then adjust parameters."
  )
```

The `file` object has `gWidgets` arguments passed through to filter which files are displayed. Otherwise, this is a straightforward mapping of arguments to type. The last three are there for the GUIs usage.

The view separates out the main arguments from some secondary ones, using the notebook container; and provides a place for the assignment and preview features. Above, we manually set the size in the `output` object above, as it does not gracefully allocate enough size to itself when the `gWidgetsRGtk2` backend is used.

```
view <- aGroup(aNotebook(
                    aNotebookPage("file",
                                  separatorItem(),
                                  "header", "sep","quote",
                                  "dec", "fill", "comment.char",
                                  label="Main args"),
                    aNotebookPage("as.is","na.strings","nrows","skip",
                                  "check.names","fill","strip.white","blank.lines.skip",
                                  "allowEscapes","stringsAsFactors",
                                  separatorItem(),
                                  "fileEncoding", "encoding",
                                  label="Extra args")
                    ),
               aContainer("assign.to",
                          aFrame("output", label="Preview")
                          ),
               horizontal=TRUE)
```

The main function called to update the UI is this one, which checks the file type, then reads in the file using either `read.csv` or `read.table`. If successful, it returns the data frame.

```
dlg$read_file <- function(., file.type, output, assign.to, ...) {
  if(file.type != "") {
    out <- try(do.call(sprintf("read.%s",file.type), list(...)), silent=TRUE)
    if(inherits(out, "try-error")) {
      cat("Error reading file of type,", file.type, "\n")
      out <- data.frame(V1="")
    }
  } else {
    out <- data.frame(V1="")
  }
  return(out)
}
```

To integrate the `read_file` method, we set up the following handler for control:

```
dlg$model_value_changed <- function(.) {
  fname <- .$get_file()
  if(file.exists(fname)) {
    for(i in c("txt","csv")) {
      if(grepl(paste("\\.",i,sep=""), fname))
        .$set_file.type(c(txt="table",csv="csv")[i])
    }
  }
  switch(.$get_file.type(),
         "csv"={.$set_sep(","); .$set_quote('\"')},
         "table"={},
         {}
         )
  .$set_output(.$do_call("read_file",.$to_R()))
}
```

This method sets up a call to our `read_file` using the values found from the `to_R` method. The use of `do_call` is identical to the familiar `do.call` function, except it gracefully handles the instance where the method does not exist.

This method is called whenever a model value is changed, even the `output` value, say, which is not really necessary. If this inefficiency bothers you, a different manner is presented at the end.

Finally, we change the default `OK_handler` to output the value to the given name. We use a `gWidgets` call to `gconfirm` to avoid overwriting a variable name without permission.

```
dlg$OK_handler <- function(.) {
  out <- .$do_call("read_file",.$to_R())
  assign.to <- .$get_assign.to()
  if(exists(assign.to, envir=.GlobalEnv)) {
    if(!gconfirm(sprintf("Overwrite variable %s?", assign.to)))
      return()
```

```
  }
  assign(assign.to, out, envir=.GlobalEnv)
}
```

The GUI is drawn as usual now:

```
dlg$make_gui(gui_layout=view)
```

To see one way to get the dialog to update only in response to the appropriate changing of the model, we could replace the `model_value_changed` code with the following. First we have a different response when the file is changed, as we need to find the file type and adjust some arguments for CSV files.

```
dlg$property_file_value_changed <- function(., value, old_value) {
  if(file.exists(value)) {
    for(i in c("txt","csv")) {
      if(grepl(paste("\\.",i,sep=""), value))
        .$set_file.type(c(txt="table",csv="csv")[i])
    }
  }
  switch(.$get_file.type(),
         "csv"={.$set_sep(","); .$set_quote('\"')},
         "table"={.$set_output(.$do_call("read_file",.$to_R()))},
         {}
         )

}
```

The call to `set_output` will be done by both `set_sep` and `set_quote` as well, so we don't repeat more than we have to.

Now to update the preview when an argument is changed, we set up the appropriate `property_XXX_value_changed` methods, as follows:

```
nms <- names(dlg$get_items())
nms <- setdiff(nms, c("file","file.type","assign.to","output"))
QT <- sapply(nms, function(i) {
  assign(sprintf("property_%s_value_changed",i),
         function(., ...) {
           .$set_output(.$do_call("read_file",.$to_R()))
         },
         envir=dlg)
})
```

The assignment of the method within the environment takes advantage of `proto` objects being environments.

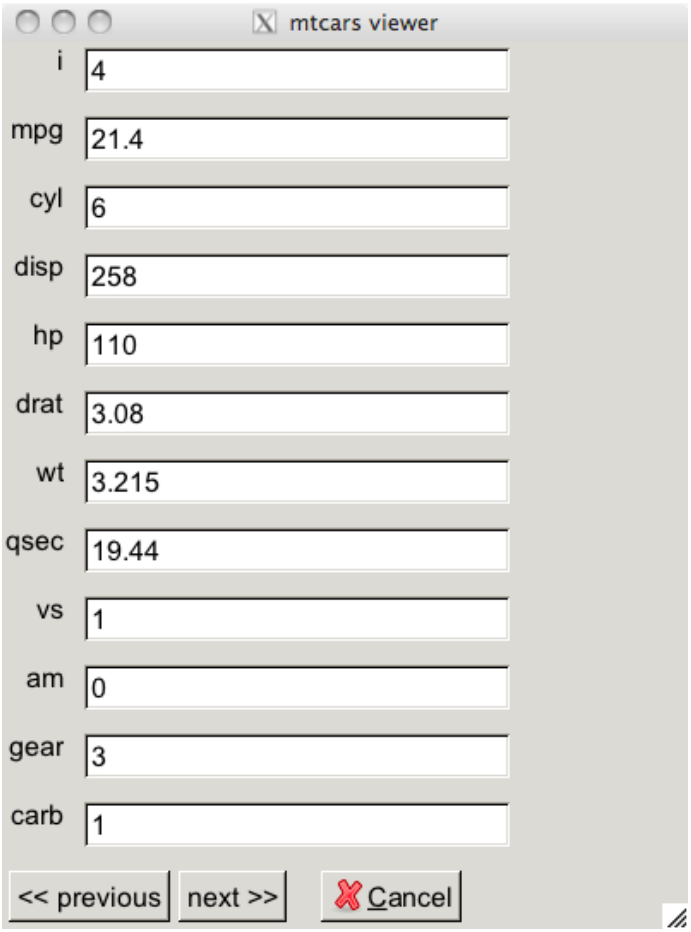

Figure 8: A GUI to scroll through the rows of a data frame.

6.4 A view data frame by row GUI

A common GUI for viewing data on web sites basically shows each row of a data frame as editable data. In statistics terms, we scroll through the cases in a data set. Below, we leave the editable part as an exercise, but show how the method `set_model` can be used to effect this type of GUI.

This function creates a list of items from a row in the data frame. The `set_model` method for a dialog takes a dialog or item group for an argument. The output of this function will provide the items value.

```
m <- mtcars
nms <- names(m)
make_model <- function(i) {
  l <- list(i = integerItem(i))
  for(j in 1:ncol(m)) {
    l[[nms[j]]] <- numericItem(m[i,j])
  }
  l
}
```

Here we initialize the dialog and make two buttons to scroll through the rows. The `set_row` method finds the next row to show, then creates an item group which is used for our new model. The same extractor method, `get_i` is used in the button handlers.

```
dlg <- aDialog(items=make_model(1),
               title="Data frame scroller",
               help_string="Press buttons to scroll through data set",
               buttons=c("<< previous","next >>","SPACE","Cancel"),
               set_row=function(.,i) {
                 if(i < 1)
                   i <- nrow(m)
                 if(i > nrow(m))
                   i <- 1
                 ig <- make_model(as.numeric(i))
                 .$set_model(anItemGroup(items=ig))
               },
               previous_handler=function(.) {
                 i <- as.integer(.$get_i())
                 .$set_row(i-1)
               },
               next_handler=function(.) {
                 i <- .$to_R()$i        # same but different
                 .$set_row(i + 1)
               })
```

We now make the GUI (Figure 8).

```
dlg$make_gui()
```

That's it. We leave for later perhaps using a scroller to scroll through the rows, or a means to edit the values and store in the data frame.

6.5 A Wizard GUI

A wizard, or multi-page dialog is a common GUI arrangement. Here is one way to implement such a feature, maybe not the best. We create a model to store our values and use instances of this model and views to show just part.

```
model <- aDialog(items=list(
                   a=stringItem(""),
                   b=stringItem("")
                  )
                  )
dlg1 <- aDialog(buttons="Next",
                Next_handler=function(.) {
                  dlg2$make_gui(gui_layout=view2)
                  .$close_gui()
                })
view1 <- aContainer("a", context=model)
dlg2 <- aDialog(buttons = c("Finished"),
                Finished_handler = function(.) {
                  print(model$to_R())
                  .$close_gui()
                })
view2 <- aContainer("b", context=model)
dlg1$make_gui(gui_layout=view1)
```