Table of Contents

1 Introduction

Most application programs interact with users through a graphical user interface. To achieve this, programmers use toolkits such as Motif or class libraries such as Swing or MFC. While Icon's platform-independent graphics facilities are excellent for drawing to the screen, manipulating windows, and so on, the standard elements of graphical user interfaces are not built-in to the language. A library of procedures called the vidgets library provides Motif-like buttons, menus, scroll bars, and so on. The vidgets library suffers from two drawbacks: it is missing many interface components that are used in modern applications, and it is not easily extended to include new components.

This chapter presents an alternative user interface class library that addresses these drawbacks. An object-oriented design is used to reduce complexity and increase the extensibility of the toolkit. The result is an elegant library, simply called "the GUI toolkit," that is accessed by importing the package `gui`. The GUI toolkit is a class library, and as such it is specific to Unicon, and not intended to address the needs of Arizona Icon programmers, who can use the vidgets library. Despite being only around two-thirds of the size (in lines) of the vidgets library, the GUI toolkit offers many more interface components. Although this book describes the components provided by the GUI toolkit, you will need the book *Graphics Programming in Icon* [Griswold98] to develop custom user interfaces with application-specific graphics.

To examine the capabilities of the GUI classes, the chapter presents an example program that allows you to design an interface by visually creating a dialog on the screen interactively. The program then generates an Unicon program that can be filled in to create a working application. When you finish this chapter you will know how to:

- Construct programs that employ a graphical user interface.
- Manipulate the attributes of objects such as buttons and scrollbars.
- Draw a program's interface using ivib, Unicon's improved visual interface builder.

2 A Simple Dialog Example

Object-orientation seems to be a big help in designing graphical user interfaces, and this is as true for Unicon as it is for other languages. The best way to see how the GUI classes work is to try out a simple example program. Listing 17-1 shows the source code in full, which is described in detail below.

Listing 1
The TestDialog Program

```
import gui
$include "guih.icn"

class TestDialog : Dialog()
   method component_setup()
      local l, b
      l := Label("label=Click to close", "pos=50%,33%",
                 "align=c,c")
      add(l)

      b := TextButton("label=Button", "pos=50%,66%", "align=c,c")
      b.connect(self, "dispose", ACTION_EVENT)
      add(b)

      attrib("size=215,150", "bg=light gray","font=serif",
             "resize=on")
   end
end

procedure main()
   local d

   d := TestDialog()
   d.show_modal()
end
```

Assuming the program is stored in a file called `testdialog.icn`, issue the following command to compile it:

```
unicon testdialog
```

The result should be an executable file called `testdialog`. For convenience, this example program, together with several others, can be found in the `guidemos` directory of the Unicon distribution. Run this program, and the window shown in Figure 17-1 should appear, and should close when the button is clicked.

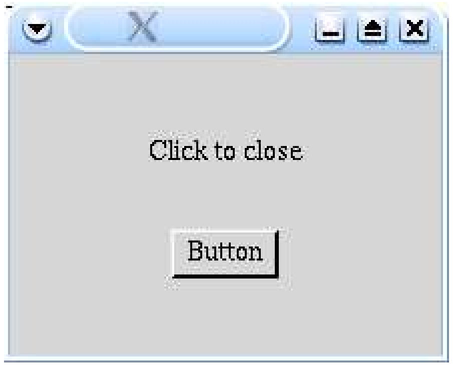


Figure 1TestDialog window

It is worth examining this example program line by line. It begins by declaring a new class, `TestDialog`. The line

```
class TestDialog : Dialog()
```

indicates that `TestDialog` is a subclass of the class `Dialog` that is defined in the toolkit. This subclass relationship is true of all dialog windows.

The remainder of the class's code is contained in a single method, `component_setup()`. This method is invoked by the toolkit, and is provided as a convenient place to setup the dialog's content.

The first thing inside the `component_setup()` method is the code that adds the label to the dialog:

```
l := Label("label=Click to close","pos=50%,33%", "align=c,c")
add(l)
```

This assigns the variable `l` to a new `Label` object and sets the label string. The horizontal position is set to 50 percent of the window width and the vertical to 33 percent of the window height. The alignment of the object is specified as centered both vertically and horizontally about the position. Finally, the label is added to the dialog with the line `add(l)`.

The code to add a button is very similar, but a `TextButton` object is created rather than a `Label` object, and the vertical position is 66 percent of the window height.

The next line is more interesting :-

```
b.connect(self, "dispose", ACTION_EVENT)
```

This adds a *listener* to the button, and tells the toolkit than whenever the button fires an `ACTION_EVENT`, which it will when it is pressed, the `dispose()` method in the class `self`, should be invoked. `self` of course refers to the `TestDialog` class, and `dispose()`is a method inherited from the base class `Dialog`, which simply closes the window. So all this means is that when the button is pressed, the dialog will close.

The next line sets the attributes of the dialog window, including its initial size. Try changing these values to experiment with other dialog styles.

```
attrib("size=215,150", "bg=light gray",
            "font=serif", "resize=on")
```

After the class comes a standard Icon `main` method. This simply creates an instance of the dialog and invokes the method

```
d.show_modal()
```

This actually displays the dialog window and goes into the toolkit's event handling loop.

3 Positioning Objects

The button and the label were positioned in the example above by specifying percentages of the window size. An object can also be positioned by giving an absolute position, or by giving a percentage plus or minus an offset. So the following are all valid position specifiers:

```
"100"
"10%"
"25%+10"
"33%-10"
```

Positions are often specified in class constructor strings, with x and y values separated by commas. By default, the position specified relates to the top left corner of the object concerned. You can use the `"align"` attribute, which takes two alignment specifiers, to change this default. The first alignment specifier is an `"l"`, `"c"`, or `"r"`, for left, center, or right horizontal alignment, respectively; the second is a "t", "c", or "b", for top, center, or bottom vertical alignment, respectively. There is one further attribute, "size", whose specifiers take the same format as the position attribute. Most of the toolkit objects default to sensible sizes, and the size attribute can often be omitted. For example, a button's size will default to a size based on the font in use and the string in the button.

A generic method `attrib(attribs...)` in a utility class `SetFields` implements attribute processing, but it mostly farms out the work to special-purpose methods that can be called directly: `set_pos(x,y)`, `set_label(s)`, `set_align(horizontal, vertical)`, and `set_size(w, h)`. Other than these setter methods, Icon graphics attributes can be interspersed. For example, the attribute "bg=green" will set the object's background color.

Here are some examples of position, alignment, and size parameters, and a description of their meaning. In the call

```
attrib("pos=50%,100", "align=c,t", "size=80%,200")
```

the object is centered horizontally in the window, and takes up 80 percent of the width; vertically its top edge starts at 100 and its height is 200 pixels. In contrast, the code

```
attrib("pos=100%,100%", "align=r,b", "size=50%,50%")
```

specifies that the object fills up the bottom right quarter of the window. The call

```
attrib("pos=33%+20,0%", "size=100,100%")
```

directs that the object's left hand side is at one-third of the window size plus 20 pixels; it is 100 pixels wide. It fills the whole window vertically.

4 A More Complex Dialog Example

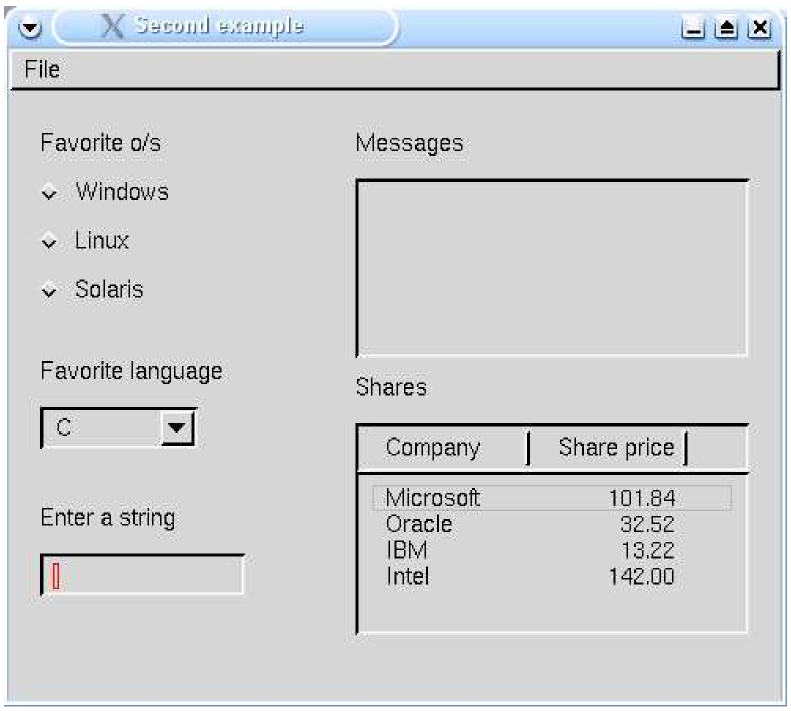
Now it's time to introduce some more component types. Listing 17-2 shows our next example program in full.

Listing 2
SecondTest Program

```
import gui
$include "guih.icn"

#
# Second test program
#
class SecondTest : Dialog(
   #
   # The class variables; each represents an object
   # in the dialog.
   #
   text_list,
   table,
   list,
   text_field,
   #
   # Some data variables.
   #
   oses,
   languages,
   shares
   )

   #
   # Add a line to the end of the text list
   #
   method put_line(s)
      local l
      l := text_list.get_contents()
      put(l, s)
      text_list.set_contents(l)
      text_list.goto_pos(*l)
   end

   #
   # Event handlers - produce a line of interest.
   #

   method handle_check_box_1(ev)
      put_line("Favourite o/s is " || oses[1])
   end

   method handle_check_box_2(ev)
      put_line("Favourite o/s is " || oses[2])
   end

   method handle_check_box_3(ev)
      put_line("Favourite o/s is " || oses[3])
   end

   method handle_text_field(ev)
      put_line("Contents = " || text_field.get_contents())
   end

   method handle_list(ev)
      put_line("Favourite language is " ||
               languages[list.get_selection()])
   end

   method handle_text_menu_item_2(ev)
      put_line("You selected the menu item")
   end

   #
   # The quit menu item
   #
   method handle_text_menu_item_1(ev)
      dispose()
   end

   method handle_table(ev)
      local i
      i := table.get_selections()[1]
      put_line(shares[i][1] || " is trading at " || shares[i][2])
   end
```

```
method handle_table_column_1(ev)
   put_line("Clicked on column 1")
end

method handle_table_column_2(ev)
   put_line("Clicked on column 2")
end

#
# This method is invoked for a component which may
# potentially want to handle an event (by firing an
# event to its listeners for example).  A dialog
# is just another custom component, and so it can
# override this method to do any custom processing.
#
method handle_event(ev)
   put_line("Icon event " || ev)
   self.Dialog.handle_event(ev)
end

method component_setup()
   local menu_bar, menu, panel_1, panel_2, panel_3,
         panel_4, panel_5, label_1, label_2, label_3,
         label_4, label_5, text_menu_item_1,
         text_menu_item_2, check_box_1, check_box_2,
         check_box_3,table_column_1, table_column_2,
         check_box_group

   #
   # Initialize some data for the objects.
   #
   oses := ["Windows", "Linux", "Solaris"]
   languages := ["C", "C++", "Java", "Icon"]
   shares := [["Microsoft", "101.84"], ["Oracle", "32.52"],
              ["IBM", "13.22"], ["Intel", "142.00"]]

   #
   # Set the attribs
   #
   attrib("size=490,400", "min_size=490,400", "font=sans",
          "bg=light gray","label=Second example", "resize=on")

   #
   # Set up a simple menu system
   #
   menu_bar := MenuBar()
   menu := Menu("label=File")
   text_menu_item_1 := TextMenuItem("label=Quit")
   text_menu_item_1.connect(self, "handle_text_menu_item_1",
                            ACTION_EVENT)
   menu.add(text_menu_item_1)
   text_menu_item_2 := TextMenuItem("label=Message")
   text_menu_item_2.connect(self, "handle_text_menu_item_2",
                            ACTION_EVENT)
   menu.add(text_menu_item_2)
   menu_bar.add(menu)
   add(menu_bar)
```

```
#
# Set-up the checkbox panel
#
check_box_group := CheckBoxGroup()
panel_1 := Panel("pos=20,50", "size=130,130")
label_2 := Label("pos=0,0", "internal_alignment=l",
                 "label=Favorite o/s")
panel_1.add(label_2)
check_box_1 := CheckBox("pos=0,30")
check_box_1.set_label(oses[1])
check_box_1.connect(self, "handle_check_box_1",
                    ACTION_EVENT)
check_box_group.add(check_box_1)
panel_1.add(check_box_1)
check_box_2 := CheckBox("pos=0,60")
check_box_2.set_label(oses[2])
check_box_group.add(check_box_2)
check_box_2.connect(self, "handle_check_box_2",
                    ACTION_EVENT)
panel_1.add(check_box_2)
check_box_3 := CheckBox("pos=0,90")
check_box_3.set_label(oses[3])
check_box_group.add(check_box_3)
check_box_3.connect(self, "handle_check_box_3",
                    ACTION_EVENT)
panel_1.add(check_box_3)
add(panel_1)

#
# The text-list of messages.
#
panel_2 := Panel("pos=220,50", "size=100%-240,50%-60")
label_1 := Label("pos=0,0", "internal_alignment=l",
                 "label=Messages")
panel_2.add(label_1)
text_list := TextDisplay("pos=0,30", "size=100%,100%-30")
text_list.set_contents([])
panel_2.add(text_list)
add(panel_2)

#
# The table of shares.
#
panel_3 := Panel("pos=220,50%","size=100%-240,50%-40")
table := Table("pos=0,30","size=100%,100%-30",
               "select_one")
table.connect(self, "handle_table",
              SELECTION_CHANGED_EVENT)
table.set_contents(shares)

table_column_1 := TableColumn("label=Company",
                              "internal_alignment=l",
                              "column_width=100")
table_column_1.connect(self, "handle_table_column_1",
                       ACTION_EVENT)
table.add_column(table_column_1)
table_column_2 := TableColumn("label=Share price",
                              "internal_alignment=r",
                                     "column_width=100")
      table_column_2.connect(self, "handle_table_column_2",
                             ACTION_EVENT)
      table.add_column(table_column_2)
      panel_3.add(table)
      label_5 := Label("pos=0,0", "internal_alignment=l",
                       "label=Shares")
      panel_3.add(label_5)
      add(panel_3)

      #
      # The drop-down list of languages.
      #
      panel_4 := Panel("pos=20,190", "size=180,50")
      list := List("pos=0,30", "size=100,")
      list.connect(self, "handle_list", SELECTION_CHANGED_EVENT)
      list.set_selection_list(languages)
      panel_4.add(list)
      label_3 := Label("pos=0,0", "internal_alignment=l",
                       "label=Favorite language")
      panel_4.add(label_3)
      add(panel_4)

      #
      # The text field.
      #
      panel_5 := Panel("pos=20,280", "size=180,50")
      label_4 := Label("pos=0,0", "internal_alignment=l",
                       "label=Enter a string")
      panel_5.add(label_4)
      text_field := TextField("pos=0,30", "size=130,",
                              "draw_border=t")
      text_field.connect(self, "handle_text_field",
                         CONTENT_CHANGED_EVENT)
      panel_5.add(text_field)
      add(panel_5)
   end
end

#
# Simple main procedure just creates the dialog.
#
procedure main()
   local d
   d := SecondTest()
   d.show_modal()
end
```



Company	Share price
Microsoft	101.84
Oracle	32.52
IBM	13.22
Intel	142.00

Figure 2SecondTest window

Start your look at this program with the `component_setup()` method at the end. This method begins by initializing some data and setting the attributes. This includes several Icon graphics attributes, as well as the minimum size attribute, which is an attribute of the `Dialog` class.

The next part creates a menu bar structure. We will deal with how menu structures work in detail later in this chapter, but for now just observe that this code creates two textual menu items within a `Menu` object, which is itself within a `MenuBar` object, which is added to the dialog. Both menu items are connected to event handler methods.

The next section sets up the three check boxes. These are placed, together with the label "Favorite o/s" in a `Panel` object. This object simply serves as a container for other objects that logically can be treated as a whole. The most important thing to note about a `Panel` is that objects within it have their size and position computed relative to the `Panel` rather than the window. So, for example, the first `Label` object is positioned with "`pos=0,0`". This places it at the top left-hand corner of the `Panel`, not the top left of the window. Percentage specifications similarly relate to the enclosing `Panel`.

Each `CheckBox` is a separate object. To make the three check boxes coordinate themselves as a group so that when one is checked another is unchecked, they are placed together in a `CheckBoxGroup` object. This does no more than "bracket" them together. When grouped together in this way the checkboxes are normally termed "radio buttons" in GUI parlance. Note that each `CheckBox` is added to the `CheckBoxGroup` *and* the `Panel`.

The next section is another `Panel` that holds another label ("Messages") and a `TextList` object. In this case the object is used to hold a list of message strings that scroll by like a terminal window. A `TextList` object can also be used for selecting one or more items from a list of strings and there is and editable version, `EditableTextList`, which can be used for editing text.

The third panel contains a label ("Shares"), and a `Table` object, which is used for displaying tabular data. Note that class `Table` has nothing to do with Icon's table data type. Adding a `TableColumn` object to the table sets up each column. The table columns have their initial column width specified with attribute `column_width` and the alignment of the column's contents set with attribute `internal_alignment`. The attribute `select_one` is used to configure the table to allow one row to be highlighted at a time. The default is not to allow highlighting of rows; the other option is to allow several to be highlighted at once with `select_many`.

The next panel contains another label ("Favorite language") and a drop-down list of selections, created using the `List` class. The selections are set using the `set_selection_list()` method. The final panel contains a label ("Enter a string") and a `TextField` object, which is used to obtain entry of a string from the keyboard.

Now note how several of the components are connected to event handlers in the class. Each handler method adds a line to the list of strings in the `TextList` by calling the `put_line()` method. The text list thus gives an idea of the events being produced by the toolkit. The exception is the menu item "Quit", which exits the program.

This dialog overrides the `handle_event()` method, which is the method invoked by the toolkit for any component (including dialogs) which may want to handle an event. In this case, the dialog just prints out the Icon event code for the particular event. This method also checks for the Alt-q keyboard combination, which closes the dialog.

5 More about event handling

As shown in the above examples, components generate events when something of interest happens. For example, a button generates an `ACTION_EVENT` when it is pressed. Different components generate different events, but there are some basic events generated by all components :-

`MOUSE_PRESS_EVENT`	Generated on a mouse press within the component's region.
`MOUSE_DRAG_EVENT`	Generated on a mouse drag within the component's region.

MOUSE_PRESS_EVENT	Generated on a mouse press within the component's region.
MOUSE_RELEASE_EVENT	Generated on a mouse release within the component's region.
MOUSE_MOTION_EVENT	Generated on a mouse motion over the component's region.

For any non-mouse events, the `Dialog` class fires an `ICON_EVENT`.

When an event occurs and is passed to a listener, an `Event` object is provided as a parameter. This object contains three fields, with corresponding getter methods, as follows :-

get_source()	Returns the component which fired the event.
get_type()	Returns the type code, eg `ICON_EVENT`
get_param()	Returns an arbitrary parameter depending on the type. However in nearly all cases this is the original underlying Icon graphics event; for example `&rrelease`.

The get_param() method is necessary, for example, to distinguish between a left mouse click and a right mouse click on a `MOUSE_RELEASE_EVENT`; for instance

```
method on_release(ev)
   if ev.get_param() === &rrelease then {
      ... process right mouse up
   }
end
```

6 Containers

Some components act as containers; that is they are components that contain other components. In fact, the `Dialog` class itself is in fact a container. We have already seen the `Panel` class in action in the last example. Now let's look at two more useful container objects in the standard toolkit: `TabSet` and `OverlaySet`.

6.1 TabSet

This class provides a container object that contains several tabbed panes, any one of which is displayed at any given time. The user switches between panes by clicking on the labeled tabs at the top of the object. The `TabSet` contains several `TabItems`, each of which contains the components for that particular pane. To illustrate this, Listing 17-3 presents a simple example of a `TabSet` that contains three `TabItems`, each of which contains a single label.

Listing 3
TabSet Program

```
import gui
$include "guih.icn"

#
# Simple example of a TabSet
#
class Tabs : Dialog(quit_button)
   method change(e)
      write("The tabset selection changed")
   end

   method component_setup()
      local tab_set, tab_item_1, tab_item_2, tab_item_3

      attrib("size=355,295", "font=sans", "bg=light gray",
             "label=TabSet example", "resize=on")

      #
      # Create the TabSet
      #
      tab_set := TabSet("pos=20,20", "size=100%-40,100%-80")

      #
      # First pane
      #
      tab_item_1 := TabItem("label=Pane 1")
      tab_item_1.add(Label("pos=50%,50%", "align=c,c",
                           "label=Label 1"))
      tab_set.add(tab_item_1)

      #
      # Second pane
      #
      tab_item_2 := TabItem("label=Pane 2")
      tab_item_2.add(Label("pos=50%,50%", "align=c,c",
                           "label=Label 2"))
      tab_set.add(tab_item_2)

      #
      # Third pane
      #
      tab_item_3 := TabItem("label=Pane 3")
      tab_item_3.add(Label("pos=50%,50%", "align=c,c",
                           "label=Label 3"))
      tab_set.add(tab_item_3)

      tab_set.set_which_one(tab_item_1)
      tab_set.connect(self, "change", SELECTION_CHANGED_EVENT)
      add(tab_set)

      #
      # Add a quit button
      #
      quit_button := TextButton("pos=50%,100%-30", "align=c,c",
                                "label=Quit")
      quit_button.connect(self, "dispose", ACTION_EVENT)
      add(quit_button)
      #
      # Close the dialog when the window close button is pressed
      #
      connect(self, "dispose", CLOSE_BUTTON_EVENT)
   end
end

procedure main()
   local d
   d := Tabs()
   d.show_modal()
end
```

The resulting window is shown in Figure 17-3:

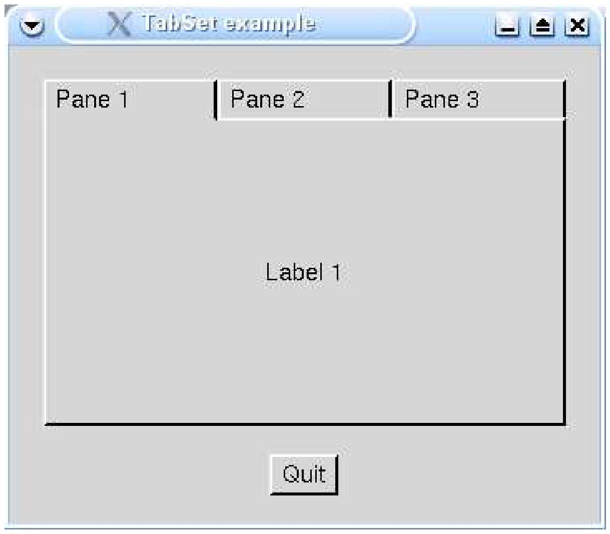


Figure 3TabSet example window

One interesting point in this dialog is the following line :-

```
      connect(self, "dispose", CLOSE_BUTTON_EVENT)
```

A dialog generates an event whenever the dialog close button is pressed. By connecting this event to the `dispose` method, the dialog is configured to close when this button is pressed.

6.2 OverlaySet

This class is very similar to the `TabSet` class, but control over which pane is currently on display is entirely under the programmer's control. There is no line of tabs to click.

Instead of adding items to `TabItem` structures, `OverlayItem` objects are used. An empty `OverlayItem` can be used if the area should be blank at some point. The current `OverlayItem` on display is set by the method `set_which_one(x)`, where `x` is the desired `OverlayItem`.

7 Menu Structures

The toolkit provides all the standard building blocks required to create a menu system (see Table 17-1). Customized components can also be added, and this is discussed later.

Table 17-1
Standard Menu System Components

Component	Description
`MenuBar`	This is the menu area along the top of the window, containing one or more `Menus`.
`MenuButton`	A floating menu bar containing one `Menu`.
`Menu`	A drop down menu pane containing other `Menus` or menu components.
`TextMenuItem`	A textual menu item.
`CheckBoxMenuItem`	A checkbox in a menu which can be part of a `CheckBoxGroup` if desired.
`MenuSeparator`	A vertical separation line between items.

Items in a `Menu` can have left and right labels as well as customized left and right images. To see how this all fits together, Listing 17-4 shows our next example program.

Listing 4
A Menu Example Program

```
import gui
$include "guih.icn"

#
# Menu example program.
#
class MenuDemo : Dialog()
   method component_setup()
      local file_menu, menu_bar, check_box_group,
            text_menu_item,labels_menu, images_menu,
            checkboxes_menu, group_menu, alone_menu,
            menu_button, check_box_menu_item, button_menu

      attrib("size=426,270", "font=sans", "bg=light gray",
             "label=Menu example")

      check_box_group := CheckBoxGroup()

      #
      # Create the menu bar.  The position and size default to
      # give a bar covering the top of the window.
      #
      menu_bar := MenuBar()

      #
      # The first menu ("File") - just contains one text item.
      #
      file_menu := Menu("label=File")
      text_menu_item := TextMenuItem("label=Quit")
      text_menu_item.connect(self, "dispose", ACTION_EVENT)
      file_menu.add(text_menu_item)
      menu_bar.add(file_menu)

      #
      # The second menu ("Labels") - add some labels
      #
      labels_menu := Menu("label=Labels")
      labels_menu.add(TextMenuItem("label=One"))
      labels_menu.add(TextMenuItem("label=Two",
                                   "label_left=ABC"))
      #
      # Add a separator and another text item
      labels_menu.add(MenuSeparator())
      labels_menu.add(TextMenuItem("label=Three",
                                   "label_right=123"))
      #
      # A sub-menu in this menu, labelled "Images"
      images_menu := Menu("label=Images")
      #
      # Add three text items with custom images.   The rather
      # unwieldy strings create a triangle, a circle and a
      # rectangle.
      #
      text_menu_item := TextMenuItem("label=One")
      text_menu_item.set_img_left("15,c1,_
~~~~~~~0~~~~~~~_
~~~~~~~0~~~~~~~_
~~~~~~000~~~~~~_
~~~~~~000~~~~~~_
~~~~~00~00~~~~~_
~~~~~00~00~~~~~_
~~~~00~~~00~~~~_
~~~~00~~~00~~~~_
~~~00~~~~~00~~~_
~~~00~~~~~00~~~_
~~00~~~~~~~00~~_
~~00~~~~~~~00~~_
~00~~~~~~~~~00~_
~0000000000000~_
000000000000000")
      images_menu.add(text_menu_item)
      text_menu_item := TextMenuItem("label=Two")
      text_menu_item.set_img_left("15,c1,_
~~~~~~000~~~~~~_
~~~~0000000~~~~_
~~000~~~~~000~~_
~~00~~~~~~~00~~_
~00~~~~~~~~~00~_
~0~~~~~~~~~~~0~_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
~0~~~~~~~~~~~0~_
~00~~~~~~~~~00~_
~~00~~~~~~~00~~_
~~000~~~~~000~~_
```

```
~~~~0000000~~~~_
~~~~~~000~~~~~~")
      images_menu.add(text_menu_item)
      text_menu_item := TextMenuItem("label=Three")
      text_menu_item.set_img_left("15,c1,_
000000000000000_
000000000000000_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
00~~~~~~~~~~~00_
000000000000000_
000000000000000")
      images_menu.add(text_menu_item)
      labels_menu.add(images_menu)
      menu_bar.add(labels_menu)

      #
      # The third menu ("Checkboxes")
      #
      checkboxes_menu := Menu("label=Checkboxes")
      #
      # Sub-menu - "Group" - two checkboxes in a checkbox group.
      #
      group_menu := Menu("label=Group")
      check_box_menu_item := CheckBoxMenuItem("label=One")
      check_box_group.add(check_box_menu_item)
      group_menu.add(check_box_menu_item)
      check_box_menu_item := CheckBoxMenuItem("label=Two")
      check_box_group.add(check_box_menu_item)
      group_menu.add(check_box_menu_item)
      checkboxes_menu.add(group_menu)
      #
      # Sub-menu - "Alone" - two checkboxes on their own
      #
      alone_menu := Menu()
      alone_menu.set_label("Alone")
      check_box_menu_item_3 := CheckBoxMenuItem("label=Three")
      alone_menu.add(check_box_menu_item_3)
      check_box_menu_item_4 := CheckBoxMenuItem("label=Four")
      alone_menu.add(check_box_menu_item_4)
      checkboxes_menu.add(alone_menu)
      menu_bar.add(checkboxes_menu)
      add(menu_bar)

      #
      # Finally, create a menu button - a mini floating menu with
      # one menu insidie it.
      #
      menu_button := MenuButton("pos=350,50%", "align=c,c")
      #
```

```
      # This is the menu, its label appears on the button.  It
      # just contains a couple of text items for illustration
      # purposes.
      #
      button_menu := Menu("label=Click")
      button_menu.add(TextMenuItem("label=One"))
      button_menu.add(TextMenuItem("label=Two"))
      menu_button.set_menu(button_menu)
      add(menu_button)
   end
end

procedure main()
   local d
   d := MenuDemo()
   d.show_modal()
end
```

The output of this program with the middle menu and its submenu open appears in Figure 17-4.

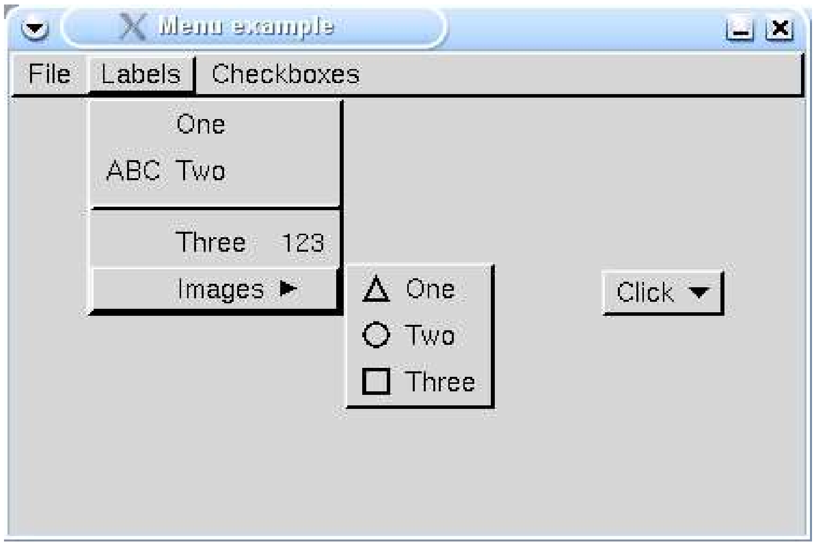


Figure 4Menus

8 Trees

The toolkit contains a tree component, which can be used to represent hierarchical data. To use it, it is necessary to create a tree-like data structure of `Node` objects. Children are added to a `Node` using its `add()` method. For example :-

```
root  := Node("label=Root")
child1  := Node("label=Child1")
child2  := Node("label=Child2")
```

```
root.add(child1)
root.add(child2)
...etc
```

After setting up the tree of `Node` objects, the root is passed to the `Tree` component for display :-

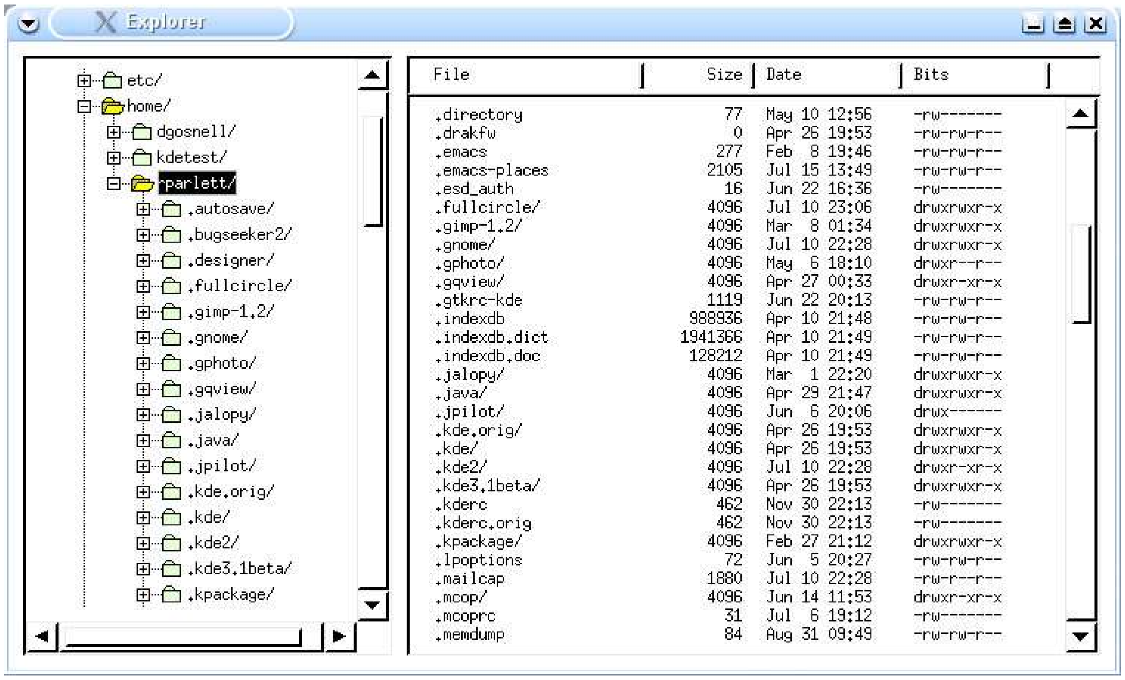
```
tree := Tree("pos=0,0", "size=100,100")
tree.set_root_node(root)
```

The tree data structure can change dynamically over time. When this occurs, the `Tree` must be notified of the change by invoking the `tree_structure_changed()` method.

The `Tree` class generates events when the selected `Node` (or `Nodes`) changes, and also when part of the tree is expanded or collapsed by the user.

Here is an example of the use of a `Tree`, together with a `Table`, to provide a filesystem explorer program.

This program also uses a `Sizer` component. This is a narrow area between the tree and the table which can be dragged to resize both dynamically. Because of the toolkit's relatively simple layout mechanism, the resizing code in `handle_sizer()` is quite awkward.



File	Size	Date	Bits
.directory	77	May 10 12:56	-rw-------
.drakfw	0	Apr 26 19:53	-rw-rw-r--
.emacs	277	Feb 8 19:46	-rw-rw-r--
.emacs-places	2105	Jul 15 13:49	-rw-rw-r--
.esd_auth	16	Jun 22 16:36	-rw-------
.fullcircle/	4096	Jul 10 23:06	drwxrwxr-x
.gimp-1.2/	4096	Mar 8 01:34	drwxrwxr-x
.gnome/	4096	Jul 10 22:28	drwxrwxr-x
.gphoto/	4096	May 6 18:10	drwxr--r--
.gqview/	4096	Apr 27 00:33	drwxr-xr-x
.gtkrc-kde	1119	Jun 22 20:13	-rw-rw-r--
.indexdb	988936	Apr 10 21:48	-rw-rw-r--
.indexdb.dict	1941366	Apr 10 21:49	-rw-rw-r--
.indexdb.doc	128212	Apr 10 21:49	-rw-rw-r--
.jalopy/	4096	Mar 1 22:20	drwxrwxr-x
.java/	4096	Apr 29 21:47	drwxrwxr-x
.jpilot/	4096	Jun 6 20:06	drwx------
.kde.orig/	4096	Apr 26 19:53	drwxrwxr-x
.kde/	4096	Apr 26 19:53	drwxrwxr-x
.kde2/	4096	Jul 10 22:28	drwxr-xr-x
.kde3.1beta/	4096	Apr 26 19:53	drwxrwxr-x
.kderc	462	Nov 30 22:13	-rw-------
.kderc.orig	462	Nov 30 22:13	-rw-------
.kpackage/	4096	Feb 27 21:12	drwxrwxr-x
.lpoptions	72	Jun 5 20:27	-rw-rw-r--
.mailcap	1880	Jul 10 22:28	-rw-r--r--
.mcop/	4096	Jun 14 11:53	drwxr-xr-x
.mcoprc	31	Jul 6 19:12	-rw-------
.memdump	84	Aug 31 09:49	-rw-rw-r--

Figure 5Explorer-style window

Listing 5
The Explorer Program

```
import gui

$include "keysyms.icn"
```

```
$include "guih.icn"

#
# A very simple filesystem explorer with a tree and a table.
#
class Explorer : Dialog(tree,
                        sizer,
                        tbl)

   #
   # Given a Node n, get the full file path it represents by
   # traversing up the tree structure to the root.
   #
   method get_full_path(n)
      local s

      s := ""
      repeat {
         s := n.get_label() || s
         n := n.get_parent_node() | break
      }
      return s
   end

   #
   # Invoked when a sub-tree is expanded (ie: the little + is
   # clicked).  An expansion event also includes contractions
   # too.
   #
   method handle_tree_expansion()
      local n

      n := tree.get_last_expanded()
      #
      # Check whether it was an expansion or a contraction.  If
      # an expansion, load the subtree and refresh the tree.
      #
      if n.is_expanded() then {
         load_subtree(n)
         tree.tree_structure_changed()
      }
   end

   #
   # Invoked when a row in the tree is selected (or de-selected).
   #
   method handle_tree_selection()
      local n

      #
      # If we have something selected, load the table.  We may
      # not have something selected if the user contracted the
      # parent node of the selected node.
      #
      if n := tree.object_get_selections()[1] then {
         load_table(n)
      }
   end
```

```
#
# Given a Node n, load its children with the sub-directories.
#
method load_subtree(n)
   local s, name, r1, dir_list, file_list

   s := get_full_path(n)

   l := get_directory_list(s)

   n.clear_children()
   every name := !l[1] do {
      if (name ~== "./") & (name ~== "../") then {
         r1 := Node("always_expandable=t")
         r1.set_label(name)
         n.add(r1)
      }
   }
end

#
# Given a Node n, load the table with the sub-files and
# sub-directories.
#
method load_table(n)
   local s, l, t

   s := get_full_path(n)

   t := get_directory_list(s)

   l := []
   every el := !sort(t[1] ||| t[2]) do {
      p := stat(s || el) | stop("No stat")
      put(l, [el, p.size, ctime(p.mtime)[5:17], p.mode])
   }

   tbl.set_contents(l)
   tbl.goto_pos(1, 0)
end

#
# The sizer has moved, so reset the sizes and positions of the
# table, tree and sizer.  Then call resize() to reposition
# everything.
#
method handle_sizer(ev)
   result_x := sizer.get_curr_pos()
   tree.set_size(result_x - 10, tree.h_spec)
   sizer.set_pos(result_x, sizer.y_spec)
   tbl.set_pos(result_x + 10, tbl.y_spec)
   tbl.set_size("100%-" || string(result_x + 20), tbl.h_spec)
   resize()
end

#
# Catch Alt-q to close the dialog.
```

```
   #
   method quit_check(ev)
      if ev.get_param() === "q" & &meta then
         dispose()
   end

   #
   # Override resize to set the sizer's min/max locations.
   #
   method resize()
      self.Dialog.resize()
      sizer.set_min_max(135, get_w_reference() - 160)
   end

   method component_setup()
      local root_node

      attrib("size=750,440", "resize=on", "label=Explorer")
      connect(self, "dispose", CLOSE_BUTTON_EVENT)
      connect(self, "quit_check", ICON_EVENT)

      tree := Tree("pos=10,10", "size=250,100%-20", "select_one")
      tree.connect(self, "handle_tree_expansion",
                   TREE_NODE_EXPANSION_EVENT)
      tree.connect(self, "handle_tree_selection",
                   SELECTION_CHANGED_EVENT)
      add(tree)

      tbl := Table("pos=270,10", "size=100%-280,100%-20",
                   "select_none")
      tbl.add_column(TableColumn("label=File",
                                 "column_width=150"))
      tbl.add_column(TableColumn("label=Size", "column_width=75",
                                 "internal_alignment=r"))
      tbl.add_column(TableColumn("label=Date",
                                 "column_width=100"))
      tbl.add_column(TableColumn("label=Bits",
                                 "column_width=100"))
      add(tbl)

      sizer := Sizer("pos=260,10", "size=10,100%-20")
      sizer.connect(self, "handle_sizer", SIZER_RELEASED_EVENT)
      add(sizer)

      #
      # Initialize the tree data structure.
      #
      root_node := Node("label=/")
      load_subtree(root_node)
      tree.set_root_node(root_node)
      tree.object_set_selections([root_node])
      load_table(root_node)
   end
end

procedure main()
   local d
   d := Explorer()
   d.show_modal()
end
```

9 Other Components

This section gives examples of some components that have not yet been encountered. For full details of how to use these classes, and the available methods and options, please see the auto-generated API documentation.

9.1 Borders

This class provides decorative borders. Optionally, a single other component can be the title of the `Border`. This would normally be a `Label` object, but it could also be a `CheckBox` or an `Icon`, or whatever is desired. The `set_title(c)` method is used to set the title. Here is a program fragment to create a border with a label as its title:

```
b := Border()
#
# Add a Label as the title
#
l := Label("label=Title String")
b.set_title(l)
add(b)
```

The `Border` class acts as a container (like a `Panel`), and so objects may be placed within it in the same way.

9.2 Images and Icons

The toolkit provides support for both images loaded from GIF files and bitmap icons defined by Icon strings. GIF files are manipulated using the Image class. The method `set_filename(x)` is used to set the location of the file to be displayed. The image will be scaled down to the size of the object, and optionally may be scaled up if it is smaller. A border may be used if desired.

Icons are created using the `Icon` class. The icon string is set using the `set_img()` method; again a border can be used if desired. Finally, the `IconButton` class lets icons serve as buttons, producing events when they are clicked.

9.3 Scroll bars

Horizontal and vertical scroll bars are available with the `ScrollBar` class. Scroll bars can be used either in the conventional way, in which the button in the bar represents a page size, and the whole bar represents a total, or as a slider in which case the button simply moves over a specified range of numbers.

10 Custom Components

This section looks at how you can create customized components that can be added to dialogs. You might want to do this to have circular rather than rectangular buttons in your dialog, for example. Or perhaps you might want to add a substantial new component type in an application, such as the main grid area in a spreadsheet program.

10.1 Creating new Components

Every component is a subclass of the `Component` class. This class contains several methods and variables that can be used by a new component.

A full list of the methods and variables defined in the `Component` class is given in the auto-generated GUI API documentation. Please refer to that documentation when reading this next example, which comes from the gui package itself :- a simple "progress bar" component.

Listing 6
A ProgressBar Component

```
package gui

$include "guih.icn"

#
# A progress bar
#
class ProgressBar : Component(
   p,         # The percentage on display.
   bar_x,
   bar_y,
   bar_h,     # Maximum bar height and width.
   bar_w
   )

   method resize()
      #
      # Set a default height based on the font size.
      #
      /self.h_spec := WAttrib(self.cwin, "fheight") +
                            2 * DEFAULT_TEXT_Y_SURROUND
      #
      # Call the parent class's method (this is mandatory).
      #
      self.Component.resize()
      #
      # Set bar height and width figures - this just gives a
      # sensible border between the "bar" and the border of the
      # object.  By using these constants, a consistent
      # appearance with other objects is obtained.
      #
      bar_x := self.x + DEFAULT_TEXT_X_SURROUND
      bar_y := self.y + BORDER_WIDTH + 3
      bar_w := self.w - 2 * DEFAULT_TEXT_X_SURROUND
      bar_h := self.h - 2 * (BORDER_WIDTH + 3)
   end

   method display(buffer_flag)
      #
      # Erase and re-draw the border and bar
      #
      EraseRectangle(self.cbwin, self.x, self.y, self.w, self.h)
      DrawRaisedRectangle(self.cbwin, self.x, self.y, self.w,
                          self.h)
      FillRectangle(self.cbwin, self.bar_x, self.bar_y,
                    self.bar_w * p / 100.0, self.bar_h)
      #
      # Draw the string in reverse mode
      #
      cw := Clone(self.cbwin, "drawop=reverse")
      center_string(cw, self.x + self.w / 2, self.y +
                    self.h / 2, p || "%")
      Uncouple(cw)
      #
      # Copy from buffer to window if flag not set.
      #
      if /buffer_flag then
         CopyArea(self.cbwin, self.cwin, self.x, self.y, self.w,
                  self.h, self.x, self.y)
   end

   #
   # Get the current percentage.
   #
   method get_percentage()
      return p
   end

   #
   # Set the percentage.
   #
   method set_percentage(p)
      p <:= 0
      p >:= 100
      self.p := p
      self.invalidate()
   end

   #
   # Provide an extra attribute, "percentage"
   #
   method set_one(attr, val)
      case attr of {
         "percentage" : set_percentage(int_val(attr, val))
         default: self.Component.set_one(attr, val)
      }
   end

   initially(a[])
      self.Component.initially()
      self.set_percentage(0)
      set_fields(a)
end
```

An example of a `ProgressBar` in use is shown below.

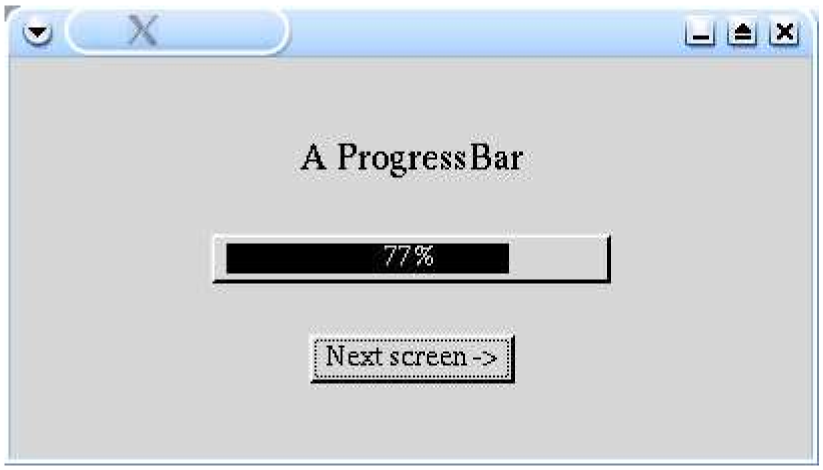


Figure 6ProgressBar in use

More complex components use other components within themselves. For example, the `TextList` class contains two `ScrollBars`, each of which in turn contains two `IconButtons`. The `Component` class has support for contained objects built in, so creating components of this sort is quite easy.

Figure 17-6 shows a dialog window containing another example component which contains an IconButton and a Label as subcomponents. A list of strings is given as an input parameter. When the button is pressed the label changes to the next item in the list, or goes back to the first one. The user can thus select any of the items.

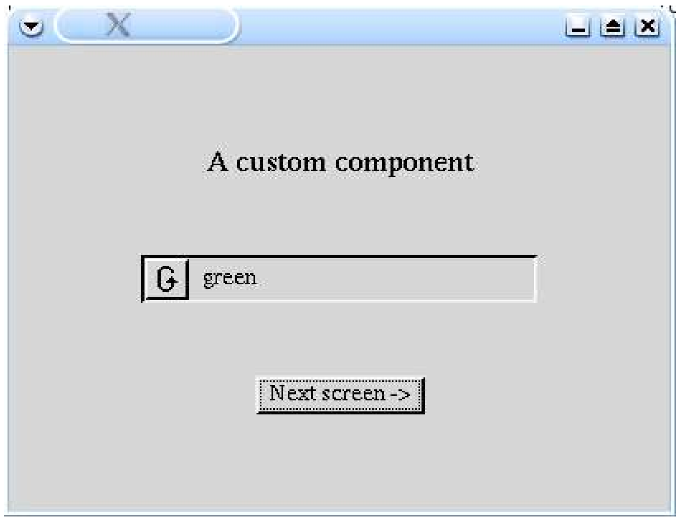


Figure 7Circulate component in use

In this example, the sub-components, `b` and `l`, are initialized in the component's constructor, and also added to the component using the `add` method. This ensures they are properly initialized. A listener is connected to the button so that the selection moves on one when it is pressed. The `set_selection()` method is available to a client program to change the selection programatically. Note how this method invokes `invalidate()`. This tells the toolkit that the component needs to be re-displayed. To keep the GUI responsive, the toolkit will only schedule this re-display when there are no user input events to be processed. So, `invalidate()` doesn't actually invoke the component's `display()` method directly.

The `resize()` method sets the sub-components' position and size, and then calls the `resize()` method for each of them. The `display()` method erases the component's rectangular area, draws a border, and draws the two sub-components into the area. The `set_one()` method is also overridden to provide some custom attributes for the component. For all other attributes, the parent class's `set_one()` method is invoked, meaning all the standard attributes work too. So, a client can construct a `Circulate` with something like :-

```
    c := Circulate("size=,250", "pos=20,20",
                   "selection_list=hot,warm,cold",
                   "selection=2", "bg=green")
```

Note how the height is omitted; the `resize()` method will set a default value.

Listing 7
Circulate component

```
package gui
link graphics

$include "guih.icn"

#
# Selection from a list
#
class Circulate : Component(selection, selection_list, b, l)
   #
   # Set the list from which selections are made.
   #
   # @param x the list of selection strings
   #
   method set_selection_list(x)
      self.selection_list := x
      self.set_selection(1)
      return x
   end

   #
   # Set the selection to the given index into the selection
   # list.
   #
   # @param x an index into the selection list
   #
   method set_selection(x)
      self.selection := x
      self.l.set_label(self.selection_list[self.selection])
      self.invalidate()
      return x
   end

   #
   # Return the current selection, as an index in the selection
   # list.
   #
   # @return an integer, being the current selection
   #
   method get_selection()
      return self.selection
   end

   #
   # Called once at startup, and whenever the window is resized.
   #
   # @p
   method resize()
      /self.h_spec := WAttrib(self.cwin, "fheight") + 16
      compute_absolutes()

      #
      # Set button position and size
      #
      b.set_pos(BORDER_WIDTH, BORDER_WIDTH)
      b.set_size(self.h - 2 * BORDER_WIDTH,
                 self.h - 2 * BORDER_WIDTH)
      b.resize()

      l.set_pos(self.h - BORDER_WIDTH + DEFAULT_TEXT_X_SURROUND,
                self.h / 2)
      l.set_align("l", "c")
      l.set_size(self.w - self.h - 2 * DEFAULT_TEXT_X_SURROUND,
                 self.h - 2 * BORDER_WIDTH)
      l.resize()

      return
   end

   #
   # Display the object.  In this case, double buffering is not
   # necessary.
   #
   # @p
   method display(buffer_flag)
      W := if /buffer_flag then self.cwin else self.cbwin
      EraseRectangle(W, self.x, self.y, self.w, self.h)
      DrawSunkenRectangle(W, self.x, self.y, self.w, self.h)
      l.display(buffer_flag)
      b.display(buffer_flag)
      self.do_shading(W)
   end

   #
   # The handler for the button - move the selection forward.
   #
```

```
   # @p
   method on_button_pressed(ev)
      self.set_selection(1 +
                         self.selection % *self.selection_list)
      create_event_and_fire(SELECTION_CHANGED_EVENT, e)
   end

   method set_one(attr, val)
      case attr of {
         "selection" : set_selection(int_val(attr, val))
         "selection_list" : set_selection_list(val)
         default: self.Component.set_one(attr, val)
      }
   end

   initially(a[])
      self.Component.initially()
      self.l := Label()
      self.l.clear_draw_border()
      add(self.l)
      self.b := IconButton()
      self.b.connect(self, "on_button_pressed", ACTION_EVENT)
      add(self.b)
      b.set_draw_border()
      self.b.set_img("13,c1,_
~~~~0000~~~~~_
~~~000000~~~~_
~~00~~~~00~~~_
~00~~~~~~00~~_
~00~~~~~~00~~_
~00~~~~~~~~~~_
~00~~~~~~~~~~_
~00~~~~~~0~~~_
~00~~~~~000~~_
~00~~~~00000~_
~00~~~0000000_
~00~~~~~~00~~_
~00~~~~~~00~~_
~00~~~~~~00~~_
~~00~~~~00~~~_
~~~000000~~~~_
~~~~0000~~~~~_
")
      set_fields(a)
end
```

11 Customized MenuComponents

Listing 17-7 contains a custom menu component. The class hierarchy for menu structures is different to other components, and so this component is a subclass of SubMenu, rather than Component. The component allows the user to select one of a number of colors from a Palette by clicking on the desired box. Again, please read this example in conjunction with the reference section on menus.

Listing 8
Color Palette Program

```
import gui

#
# The standard constants
#
$include "guih.icn"

#
# Width of one colour cell in pixels
#
$define CELL_WIDTH 30

#
# The Palette class
#
class Palette : SubMenu(
   w,                         #
   h,                         #
   colour,                    # Colour number selected
   palette,                   # List of colours
   box_size,                  # Width/height in cells
   temp_win                   # Temporary window
   )

   #
   # Get the result
   #
   method get_colour()
      return self.palette[self.colour]
   end

   #
   # Set the palette list
   #
   method set_palette(l)
      box_size := integer(sqrt(*l))
      return self.palette := l
   end

   #
   # This is called by the toolkit; it is a convenient place to
   # initialize any sizes.
   #
   method resize()
      self.w := self.h := self.box_size * CELL_WIDTH +
                2 * BORDER_WIDTH
   end

   #
   # Called to display the item.  The x, y co-ordinates have
   # been set up for us and give the top left hand corner of
   # the display.
   #
   method display()
      if /self.temp_win then {
         #
         # Open a temporary area for the menu and copy.
         #
         self.temp_win := WOpen("canvas=hidden", "size=" ||
                                self.w || "," || self.h)
         CopyArea(self.parent_component.get_parent_win(),
                  self.temp_win, self.x, self.y, self.w, self.h,
                  0, 0)
      }

      cw := Clone(self.parent_component.cwin)

      #
      # Clear area and draw rectangle around whole
      #
      EraseRectangle(cw, self.x, self.y, self.w, self.h)
      DrawRaisedRectangle(cw, self.x, self.y, self.w, self.h)

      #
      # Draw the colour grid.
      #
      y1 := self.y + BORDER_WIDTH
      e := create "fg=" || !palette
      every 1 to box_size do {
         x1 := self.x + BORDER_WIDTH
         every 1 to box_size do {
            WAttrib(cw, @e)
            FillRectangle(cw, x1, y1, CELL_WIDTH, CELL_WIDTH)
            x1 +:= CELL_WIDTH
         }
         y1 +:= CELL_WIDTH
      }
      Uncouple(cw)
   end

   #
   # Test whether pointer in palette_region, and if so which
   # cell it's in
   #
   method in_palette_region()
      if (self.x <= &x < self.x + self.w) &
         (self.y <= &y < self.y + self.h) then {
         x1 := (&x - self.x - BORDER_WIDTH) / CELL_WIDTH
         y1 := (&y - self.y - BORDER_WIDTH) / CELL_WIDTH
         return 1 + x1 + y1 * box_size
      }
   end

   #
   # Will be called if our menu is open.
   #
   method handle_event(e)
      if i := self.in_palette_region() then {
         if integer(e) = (&lrelease | &rrelease |
                          &mrelease) then {
            self.colour := i
            # This is a helper method in the superclass which
            # closes the menu system and fires an ACTION_EVENT
            succeed(e)
         }
      } else {
         if integer(e) = (&lrelease | &rrelease | &mrelease |
                          &lpress | &rpress | &mpress) then
            # This is a helper method in the superclass which
            # closes the menu system, without firing an event.
            close_all()
      }
   end

   #
   # Close this menu.
   #
   method hide()
      #
      # Restore window area.
      #
      cw := self.parent_component.cwin
      EraseRectangle(cw, self.x, self.y, self.w, self.h)
      CopyArea(self.temp_win,
               self.parent_component.get_parent_win(),
               0, 0, self.w, self.h, self.x, self.y)
      WClose(self.temp_win)
      self.temp_win := &null
   end

   # Support a "palette" attrib
   method set_one(attr, val)
      case attr of {
         "palette" : set_palette(val)
         default : self.MenuComponent.set_one(attr, val)
      }
   end

   initially(a[])
      self.SubMenu.initially()
      #
      # Set the image to appear on the Menu above ours.  We
      # could design a tiny icon and use that instead of the
      # standard arrow if we wished.
      #
      self.set_img_right(img_style("arrow_right"))
      #
      # Support the attrib style constructor.
      #
      set_fields(a)
end

#
# Test class dialog.
#
class TestPalette : Dialog(palette)
   method on_palette(ev)
      write("Colour selected : " || palette.get_colour())
   end

   method on_anything(ev)
      write("Anything item selected")
   end
   method component_setup()
      local menu_bar, menu, text_menu_item, close

      attrib("size=400,200")

      #
      # Create a MenuBar structure which includes our palette
      # as a sub-menu
      #
      menu_bar := MenuBar("pos=0,0")
      menu := Menu("label=Test")
      text_menu_item := TextMenuItem("label=Anything")
      text_menu_item.connect(self, "on_anything", ACTION_EVENT)
      menu.add(text_menu_item)

      palette := Palette("label=Test menu",
                         "palette=red,green,yellow,black,_
                          white,purple,gray,blue,pink")

      palette.connect(self, "on_palette", ACTION_EVENT)
      menu.add(palette)
      menu_bar.add(menu)
      add(menu_bar)

      #
      # Add a close button.
      #
      close := TextButton("pos=50%,66%", "align=c,c",
                          "label=Close")
      close.connect(self, "dispose", ACTION_EVENT)
      add(close)
   end
end

#
# Main program entry point.
#
procedure main()
   local d
   d := TestPalette()
   d.show_modal()
end
```

The resulting window, with the `Palette` menu active is shown in Figure 17-7.

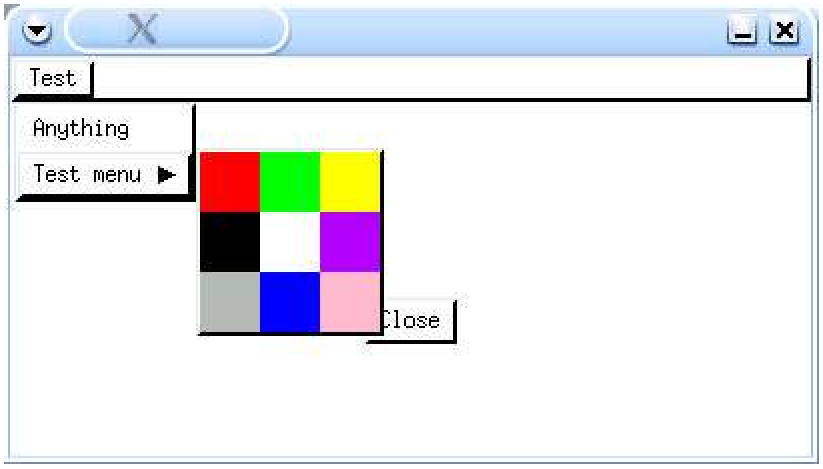


Figure 8TestPalette window

12 Keyboard accelerators

Keyboard accelerators can easily be set on any component. Simply use the method `set_accel()`, or by using the “accel” attribute. For example :-

```
b := TextButton("label=Click me", "accel=c")
```

The key combination Alt-c will then have the effect of pressing the button. For most other components, the effect of their accelerator key is to give them keyboard focus.

A label may be linked to another component in order to display its accelerator key as an underlined character in the label text, as follows :-

```
          l := Label("label=Enter some text")
          t := TextField("accel=e")
          l.set_linked_accel(t)
```

This will have the effect of underlining the “E” in the label text. When Alt-e is pressed, the text field will gain the keyboard focus.

Menu components can also have keyboard accelerators set in just the same way as components. The top-level menus in a menu bar will then respond to their accelerators by opening. Sub-items with accelerator keys can then be selected just by pressing their accelerator key (without Alt being pressed).

13 Tickers

Nearly all GUI programs spend most of their time waiting for user input events. The toolkit allows this spare time to be exploited by user programs by the use of "tickers". These are classes of type `Ticker`, which fire events to their listeners at a specified interval. In fact, if the toolkit is busy processing input events or updating the display, the actual interval may be much greater than that requested (but it will never be less).

Many components make use of tickers. For example when the mouse button is dragged below the bottom of an `EditableTextList` and held, the cursor scrolls downwards without any user events occurring. This is handled with a ticker.

Here is a simple ticker example :-

```
class SomeType()
   method on_tick()
      write("Doing something")
   end
end
...
   obj := SomeType()
   t := Ticker()
   t.connect(obj, "on_tick", TICK_EVENT)
   t.start(2000)
```

All this is doing is creating a `Ticker` instance and connecting its `TICK_EVENT` events to the `on_tick` method of `obj`. After the `start()` method is invoked, the `on_tick()` method will be invoked roughly every 2000ms.

Tickers can only be used whilst there is at least one dialog window open, because it is from within the toolkit's event processing loop that tickers are scheduled.

For convenience, the base `Component` class has some ticker-related helper methods. A `Component` subclass (which includes any subclass of `Dialog`), can simply implement a `tick()` method to use the ticker facility. It simply needs to invoke `set_ticker()` and `stop_ticker()` to start and stop a ticker, respectively. Another method, `retime_ticker()`, allows the rate of an active ticker to be changed.

One important rule regarding ticker programming has to be borne in mind, and that is that a `TICK_EVENT` handler method must return quite quickly; at most within a few tenths of a second. If it does not, then the GUI may become unresponsive to user events, which cannot be processed whilst control is in the handler method.

So, the `TICK_EVENT` method must return quickly. But what if the task you want to implement in the background is by nature deeply structured and takes a long time. In this case, it may be difficult to get out of the method promptly and continue in the same state on the next tick. Fortunately Icon has a feature which can be very helpful here, and that is co-expressions. A co-expression maintains its own stack and can suspend itself at any point, and then continue again later with the state (including stack) intact.

Here is an example program which illustrates these points. It generates prime numbers using the Erastothenes' sieve method, in a ticker, using a co-expression to conveniently suspend generation after each prime.

This program also introduces a new component, namely a slider which is used to increase or decrease the ticker rate dynamically.

Listing 9
Erastothenes' sieve Program

```
import gui
$include "guih.icn"

$define PRIME_LIMIT 20000

#
```

```
# A program to calculate prime numbers in a background ticker,
# and display them in a dialog.
#
class Sieve : Dialog(prime_ce,
                     interval,
                     count_label,
                     prime_label,
                     rate_label,
                     start,
                     stop)

   #
   # Bring the label and ticker into line with the interval
   # slider.
   #
   method synch_interval()
      rate_label.set_label(interval.get_value() || " ms")
      #
      # If the ticker is running, retime it.
      #
      if is_ticking() then
         retime_ticker(interval.get_value())
   end

   #
   # Toggle the grey state of the start/stop buttons.
   #
   method toggle_buttons()
      start.toggle_is_shaded()
      stop.toggle_is_shaded()
   end

   #
   # Called when the start button has been pressed: toggle the
   # grey state and start the ticker.
   #
   method on_start()
      toggle_buttons()
      set_ticker(interval.get_value())
   end

   #
   # Called when the stop button has been pressed: toggle the
   # grey state and stop the ticker.
   #
   method on_stop()
      toggle_buttons()
      stop_ticker()
   end

   #
   # The tick method, which is invoked regularly by the toolkit.
   # It just invokes the co-expression to display the next prime.
   #
   method tick()
      @prime_ce
   end
```

```
#
# This method consitutes the co-expression body.
#
method primes()
   local prime_candidate, non_prime_set, prime_count

   non_prime_set := set()
   prime_count := 0

   every prime_candidate := 2 to  PRIME_LIMIT do {
      if not member(non_prime_set, prime_candidate) then {
         #
         # Update the UI
         #
         count_label.set_label(prime_count +:= 1)
         prime_label.set_label(prime_candidate)
         #
         # Update the non-prime set.
         #
         every insert(non_prime_set,
                      2 * prime_candidate to PRIME_LIMIT by
                          prime_candidate)
         #
         # Suspend the co-expression until the next tick.
         #
         @&source
      }
   }
end

method component_setup()
   local prime_border, rate, buttons, b

   attrib("size=325,200", "label=Sieve")
   connect(self, "dispose", CLOSE_BUTTON_EVENT)

   prime_border := Border("pos=20,20", "size=100%-40,78")
   prime_border.set_title(Label("pos=10,0", "label=Primes"))
   prime_ce := create primes()

   prime_border.add(Label("pos=20,18", "label=Prime:"))
   count_label := Label("pos=77,18", "size=40")
   count_label.set_label("")
   prime_border.add(count_label)
   prime_border.add(Label("pos=20,40", "label=Value:"))
   prime_label := Label("pos=77,40", "size=40")
   prime_label.set_label("")
   prime_border.add(prime_label)

   add(prime_border)

   rate := Panel("pos=20,112", "size=100%-40,30")
   rate.add(Label("pos=0,50%", "size=45", "align=l,c",
                  "label=Rate:"))
   interval := Slider("pos=45,50%", "size=100%-90",
                      "align=l,c", "range=20,2020",
                      "is_horizontal=t")
   interval.set_value(1000)
```

```
      interval.connect(self, "synch_interval",
                       SLIDER_DRAGGED_EVENT)
      rate.add(interval)

      rate_label := Label("pos=100%,50%", "size=45",
                          "align=r,c", "internal_alignment=r")
      rate.add(rate_label)
      synch_interval()
      add(rate)

      buttons := Panel("pos=50%,158", "size=161,25", "align=c,t")
      start := TextButton("pos=0,0", "label=Start")
      start.connect(self, "on_start", ACTION_EVENT)
      buttons.add(start)
      stop := TextButton("pos=58,0", "label=Stop", "is_shaded=t")
      stop.connect(self, "on_stop", ACTION_EVENT)
      buttons.add(stop)
      b := TextButton("pos=108,0", "label=Quit")
      b.connect(self, "dispose", ACTION_EVENT)
      buttons.add(b)
      add(buttons)
   end
end

procedure main()
   local d
   d := Sieve()
   d.show_modal()
end
```

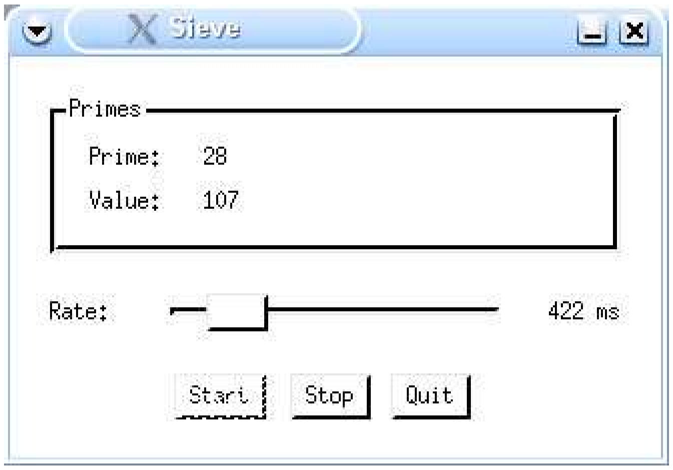


Figure 9Sieve program

14 Advanced list handling

Several of the more sophisticated components extend a common base class, `SelectableScrollArea`, na10mely `TextList`, `Table` and `Tree`. (In fact, `Table` doesn't directly extend `SelectableScrollArea` ; it contains a header component and a content component that does). It is quite easy to add some advanced features to these components, such as right-click popup menus, multi-selection and drag and drop, and this is explained in this section.

14.1 Selection

Selection handling is straightforward. First, configure the component so that it allows selection of no, one, or many rows using the methods `set_select_none()`, `set_select_one()` or `set_select_many()`, or the attributes "`select_none`", "`select_one`" or "`select_many`".

Then, listen for changes by listening for a `SELECTION_CHANGED_EVENT` to be fired :-

```
                          comp.connect(self,     "handle_selection",
SELECTION_CHANGED_EVENT)
```

When such an event does occur, the current selections can be retrieved in one of two ways. Either by getting the indexes of the selections using `get_selections()`, or by getting the objects selected, using `object_get_selections()`. The former returns a list of integers, the latter a list of objects whose type depends on the component. For a `TextList`, a list of strings is returned, for a `Table`, a list of lists (each being a row's data), and for a `Tree`, a list of `Node` objects is returned.

There are corresponding setter methods for setting the selection dynamically.

14.2 Popups

Adding popup menus is also easy. First create a `Popup` component, ready to be shown. Then, listen for a `MOUSE_RELEASE_EVENT`. Finally, when an event occurs check that is a right mouse release, and that the object is in the state you want. If it is, just activate the popup via its `popup()` method.

14.3 Drag and drop

The toolkit supports a limited form of drag and drop. The limitation is that drag and drop can only take place between components within the same window. Nonetheless, this can still prove to be a useful feature.

To implement drag and drop, a class, `DndHandler`, must be subclassed and an instance "plugged-in" to the component which is a source or target of a potential drag and drop operation, using its `set_dnd_handler()` method.

The `DndHandler` class provides five callback methods which the toolkit uses to control a drag and drop operation.

When using the `SelectableScrollArea` family of components, it is best to subclass `SelectableScrollAreaDndHandler`, which is a custom subclass of `DndHandler`, with several methods already defined appropriately.

All of the above features are brought together in the following example program which provides a `Tree` and a `TextList`. Drag and drop is enabled between the two components, and both provide popup menus for adding/deleting elements.

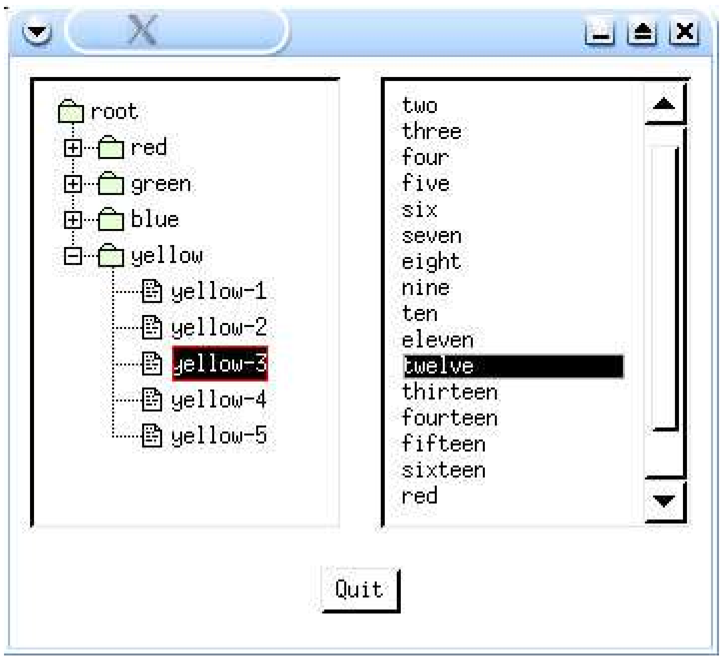


Figure 10Drag and drop

Listing 11
The Drag and drop Program

```
import gui

$include "guih.icn"

#
# A DndHandler for the list
#
class ListDndHandler : SelectableScrollAreaDndHandler()
   #
   # A drop has occurred; we succeed iff we accept it
   #
   method can_drop(d)
      local l, ll
      if l := parent.get_highlight() then {
         if d.get_source() === parent then {
            #
            # Move within the list itself
            #
            parent.move_rows(parent.get_gesture_selections(), l)
         } else {
            #
            # Copy from tree to list.  d.get_content() gives
            # a list of the nodes being dragged.
            #
            ll := []
            every el := !d.get_content() do {
               #
               # Don't drag folders.
               #
               if /el.is_folder_flag then
                  put(ll, el.get_label())
            }
            parent.insert_rows(ll, l)
         }
         return
      }
   end

   #
   # This is invoked after a successful operation when the
   # list was the source.  If the destination (c) wasn't the
   # list, then we must delete the rows from the list.
   #
   method end_drag(d, c)
      if c ~=== parent then
         parent.delete_rows(parent.get_gesture_selections())
   end
end

#
# A DndHandler for the tree
#
class TreeDndHandler : SelectableScrollAreaDndHandler()
   #
   # Called during a drag event
   #
   method drag_event(d)
      #
      # We succeed if and only if the user is dragging over a row
      # (this is handled by the parent) AND the thing we're over
      # is a folder.
      #
      if self.SelectableScrollAreaDndHandler.drag_event(d) then
         return \parent.object_get_highlight().is_folder_flag
   end

   #
   # A drop has occurred; we succeed iff we accept it
   #
   method can_drop(d)
      local s, other, n, el
      #
      # Only consider a drop on a folder
      #
      if other := parent.object_get_highlight() &
                  \other.is_folder_flag then {
         if d.get_source() === parent then {
            #
            # If parent is the drop source, then we have a dnd
            # from within the tree.  So, we just move the nodes.
            # d.get_content() will be a list of the nodes that
            # were dragged.
            #
            every el := !d.get_content() do {
               if el.get_parent_node().delete_node(el) then
                  other.add(el)
            }
         } else {
            #
            # Drop from list.  In this case d.get_content() will
            # be a list of strings.
            #
            every el := !d.get_content() do {
               n := TreeNode()
               n.set_label(el)
               other.add(n)
            }
         }

         #
         # Notify the tree that the node data structure has
         # altered.
         #
         parent.tree_structure_changed()
         return
      }
   end

   #
   # This is invoked after a successful operation when the
   # tree was the source.  If the destination (c) wasn't the
   # tree, then we must delete the nodes from the tree.
   #
   method end_drag(d, c)
      if c ~=== parent then {
         #
         # Delete all the nodes which will have been dragged.
         #
         every n := !parent.object_get_gesture_selections() do {
            if /n.is_folder_flag then {
               n.get_parent_node().delete_node(n)
            }
         }

         #
         # Notify the tree that the node data structure has
         # altered.
         #
         parent.tree_structure_changed()
      }
   end
end

#
# We use a custom Node subclass to also store an "is_folder_flag"
# flag.
#
class TreeNode : Node(is_folder_flag)
   initially
      self.Node.initially()
      if \is_folder_flag then
```

```
         set_bmps([img_style("closed_folder"), img_style
("closed_folder"), img_style("closed_folder")])
end

#
# The main dialog.
#
class DNDTest : Dialog(tree,
                       lst,
                       tree_popup,
                       list_popup,
                       new_folder_menu_item,
                       delete_node_menu_item,
                       delete_rows_menu_item)

   #
   # Delete nodes handler
   #
   method on_delete_node()
      local n, i, l

      every n := !(tree.object_get_gesture_selections()) do {
         n.get_parent_node().delete_node(n)
      }
      #
      # Notify the tree that the node data structure has altered.
      #
      tree.tree_structure_changed()
   end

   #
   # Create a new folder
   #
   method on_new_folder()
      local n, o

      #
      # Simply add a new node under the cursor, and notify the
      # tree that the data structure changed.
      #
      if o := tree.object_get_cursor() then {
         n := TreeNode(1)
         n.set_label("New folder")
         o.add(n)
         tree.tree_structure_changed()
      }
   end

   #
   # Delete rows from the list
   #
   method on_delete_rows()
      lst.delete_rows(lst.get_gesture_selections())
   end

   #
   # Add some rows to the list, at the cursor position, or at
   # the top if there is no cursor.
```

```
#
method on_new_rows()
   lst.insert_rows(["new1", "new2", "new3"],
                   lst.get_cursor() | 1)
end

#
# Helper method to create a tree structure.
#
method create_tree()
   local r, n
   r := TreeNode(1)
   r.set_label("root")

   every s := "red" | "green" | "blue" | "yellow" do {
      n := TreeNode(1)
      n.set_label(s)
      r.add(n)
      every t := 1 to 5 do {
         o := TreeNode()
         o.set_label(s || "-" ||t)
         n.add(o)
      }
   }
   return r
end

#
# A selection-up event on the tree
#
method on_tree_release(ev)
   local n

   #
   # If the Icon event was a right mouse release,
   # display the popup at the cursor.
   #
   if ev.get_param() === &rrelease then {
      n := tree.object_get_cursor() | fail
      #
      # Adjust the shading depending on the node
      # type.
      #
      if /n.is_folder_flag then
         new_folder_menu_item.set_is_shaded()
      else
         new_folder_menu_item.clear_is_shaded()
      if n === tree.get_root_node() then
         delete_node_menu_item.set_is_shaded()
      else
         delete_node_menu_item.clear_is_shaded()

      tree_popup.popup()
   }
end

#
# A mouse release event on the list
```

```
#
method on_list_release(ev)
   if ev.get_param() === &rrelease then {
      #
      # If some rows to delete...
      #
      if lst.get_gesture_selections() then
         delete_rows_menu_item.clear_is_shaded()
      else
         delete_rows_menu_item.set_is_shaded()

      list_popup.popup()
   }
end

method component_setup()
   local m, quit, mi

   attrib("size=350,295", "resize=on")
   connect(self, "dispose", CLOSE_BUTTON_EVENT)

   tree := Tree("pos=50%-10,10", "size=50%-20,100%-70",
                "align=r,t", "select_many",
                "show_root_handles=f")
   tree.set_root_node(create_tree())
   tree.set_dnd_handler(TreeDndHandler(tree))
   tree.connect(self, "on_tree_release", MOUSE_RELEASE_EVENT)

   add(tree)

   quit := TextButton("pos=50%,100%-40", "align=c,t",
                      "label=Quit")
   quit.connect(self, "dispose", ACTION_EVENT)
   add(quit)

   #
   # Create a TextList, with some arbitrary content.
   #
   lst := TextList("pos=50%+10,10", "size=50%-20,100%-70",
                   "select_many",
                   "contents=one,two,three,four,five,_
                    six,seven,eight,nine,ten,eleven,_
                    twelve,thirteen,fourteen,fifteen,_
                    sixteen,red,blue,green")

   lst.connect(self, "on_list_release", MOUSE_RELEASE_EVENT)
   lst.set_dnd_handler(ListDndHandler(lst))
   add(lst)

   tree_popup := PopupMenu()
   m := Menu()
   tree_popup.set_menu(m)

   delete_node_menu_item := TextMenuItem("label=Delete")
   delete_node_menu_item.connect(self, "on_delete_node",
                                 ACTION_EVENT)
   m.add(delete_node_menu_item)
```

```
      new_folder_menu_item := TextMenuItem("label=New folder")
      new_folder_menu_item.connect(self, "on_new_folder",
                                   ACTION_EVENT)
      m.add(new_folder_menu_item)
      add(tree_popup)

      list_popup := PopupMenu()
      m := Menu()
      list_popup.set_menu(m)
      delete_rows_menu_item := TextMenuItem("label=Delete")
      delete_rows_menu_item.connect(self, "on_delete_rows",
                                    ACTION_EVENT)
      m.add(delete_rows_menu_item)
      mi := TextMenuItem("label=Insert rows")
      mi.connect(self, "on_new_rows", ACTION_EVENT)
      m.add(mi)

      add(list_popup)
   end
end

procedure main()
   local d
   d := DNDTest()
   d.show_modal()
end
```

15 Programming techniques

Some of the earlier example dialogs were effectively "application windows." In other words, the top-level window of a program. This section looks at some techniques for integrating dialog windows that are sub-windows into a program.

15.1 Parameters

A dialog window will normally have parameters that the calling program will want to pass to it before it is displayed using the `show()` method. Possibly the attribute syntax "`key=val`" should be supported, and perhaps a default value should be set. All of these things are easily supported by following the following structure :-

```
class AnyDialog : Dialog(a_variable)
   method set_a_variable(x)
      a_variable := x
   end
   ...
   method set_one(attr, val)
      case attr of {
         "a_variable" :
            set_a_variable(string_val(attr, val))
         default: self.Dialog.set_one(attr, val)
      }
   end

   method component_setup()
      # Initialize Components, possibly depending upon
      # the value of a_variable
      ...
      # Configure the window itself...
      attrib("size=300,200")
   end

   initially(a[])
      self.Dialog.initially()
      a_variable := "default value"
      set_fields(a)
end
```

You then use the following code in the calling program:

```
   d := AnyDialog()
   d.set_a_variable("something")
```

or

```
   d := AnyDialog("a_variable=something")
```

or just

```
   d := AnyDialog()
```

to use the default for `a_variable`. Furthermore, the standard dialog atttributes can still be used as you would expect :-

```
   d := AnyDialog("a_variable=something", "font=times",
                  "bg=green", "fg=red")
```

Another nice feature of this pattern is that subclassing can follow the same pattern. For example :-

```
class AnotherDialog : AnyDialog(another_variable)
   method set_another_variable(x)
      another_variable := x
   end
   ...
   method set_one(attr, val)
      case attr of {
         "another_variable" :
            set_another_variable(string_val(attr, val))
         default: self.AnyDialog.set_one(attr, val)
      }
   end

   method component_setup()
      self.AnyDialog.component_setup()
      ...
   end

   initially(a[])
      self.AnyDialog.initially()
      another_variable := "default value"
      set_fields(a)
end
```

At first sight, it might seem that `set_fields()` will be invoked twice, which may cause problems. In fact, because we are calling the `AnyDialog` constructor with no parameters, the `set_fields()` call in that constructor has no effect. We just have to remember to call `set_fields(a)` in the `AnotherDialog` constructor itself. This will delegate its work up to the parent classes' `set_one()` methods to handle all of the possible attributes we may give it.

Getting results back out to the calling program is very easy. The dialog can just set a result variable that can be retrieved by the caller using one of the dialog's methods.

16 Ivib

Creating a dialog window with many components can be hard work, involving repeated compiles and runs to get the components correctly sized and positioned. Furthermore, much of the code in a dialog is lengthy and tiresome to write. To help reduce the work involved for the programmer in creating a dialog, a visual interface builder is available, called Ivib. This program allows a user to interactively place and configure components in a window area. A program is then generated automatically that implements the interface. Ivib owes inspiration to VIB, a program written by Mary Cameron and greatly extended by Gregg Townsend. The main window of Ivib, with a dialog under construction, is shown in Figure 17-9.

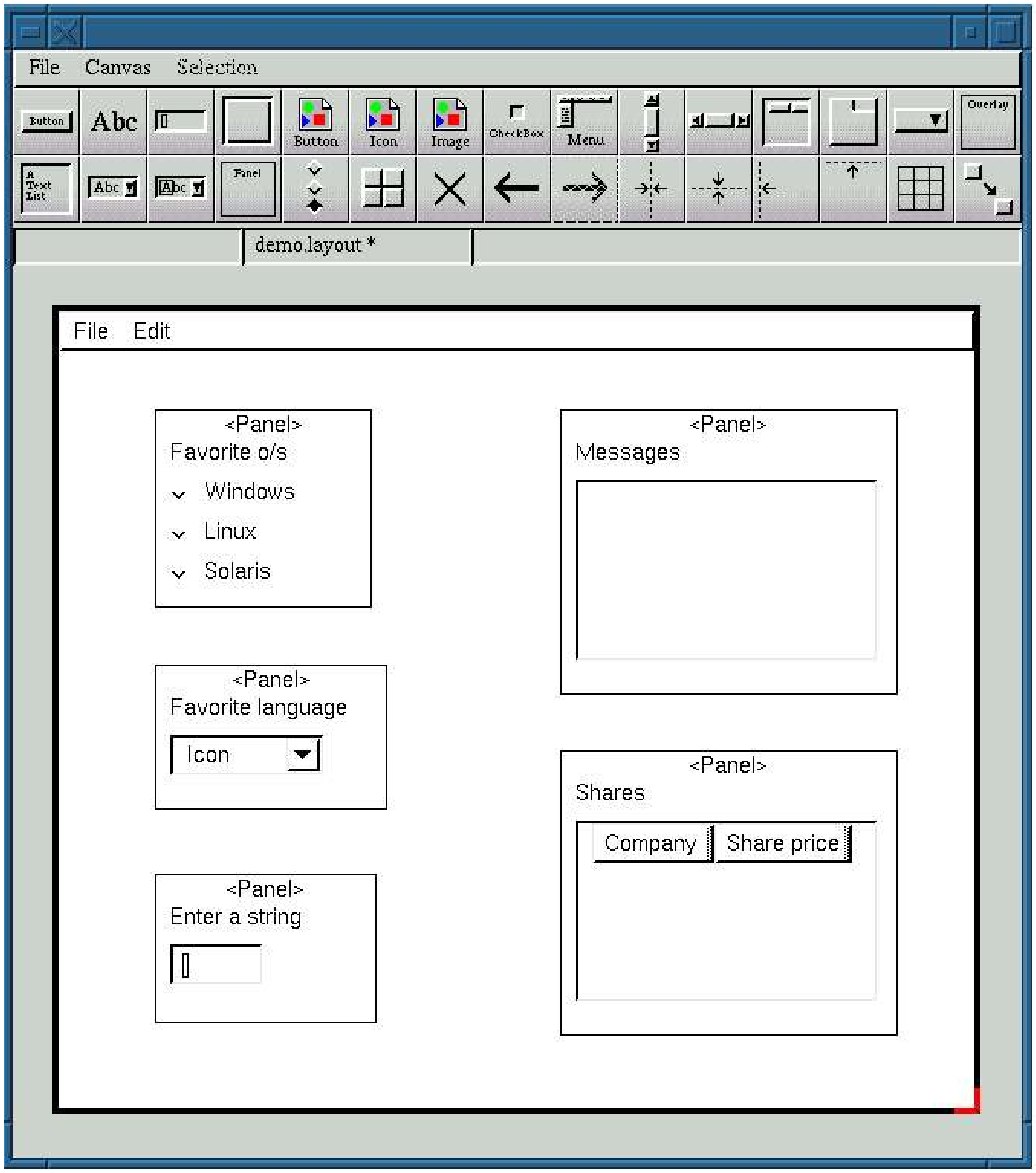
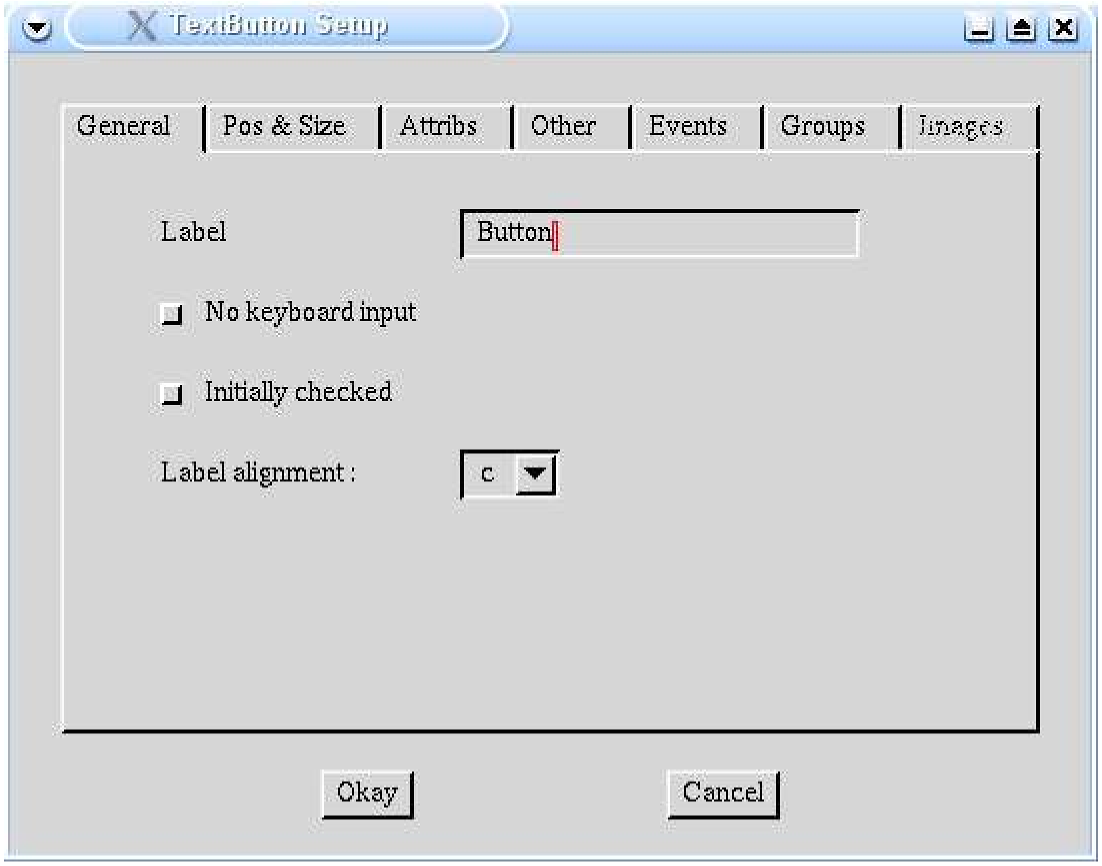


Figure 11Ivib main window

To create a dialog window using Ivib, start the program with the name of a new source file. For example:

```
ivib myprog.icn
```

The Ivib window will appear with a blank "canvas" area, which represents the dialog window to be created. At startup, the attributes of this window are the default Icon window attributes. Before you learn how to change these attributes, here is how you add a button to the dialog. Clicking the button in the top left-hand corner of the toolbar does this. Try moving and resizing the resulting button by left-clicking on it with the mouse. To change the label in the button, click on it so that the red borders appear in the edges. Then press Alt-D. The dialog shown in Figure 17-10 appears.



Label: Button

☐ No keyboard input

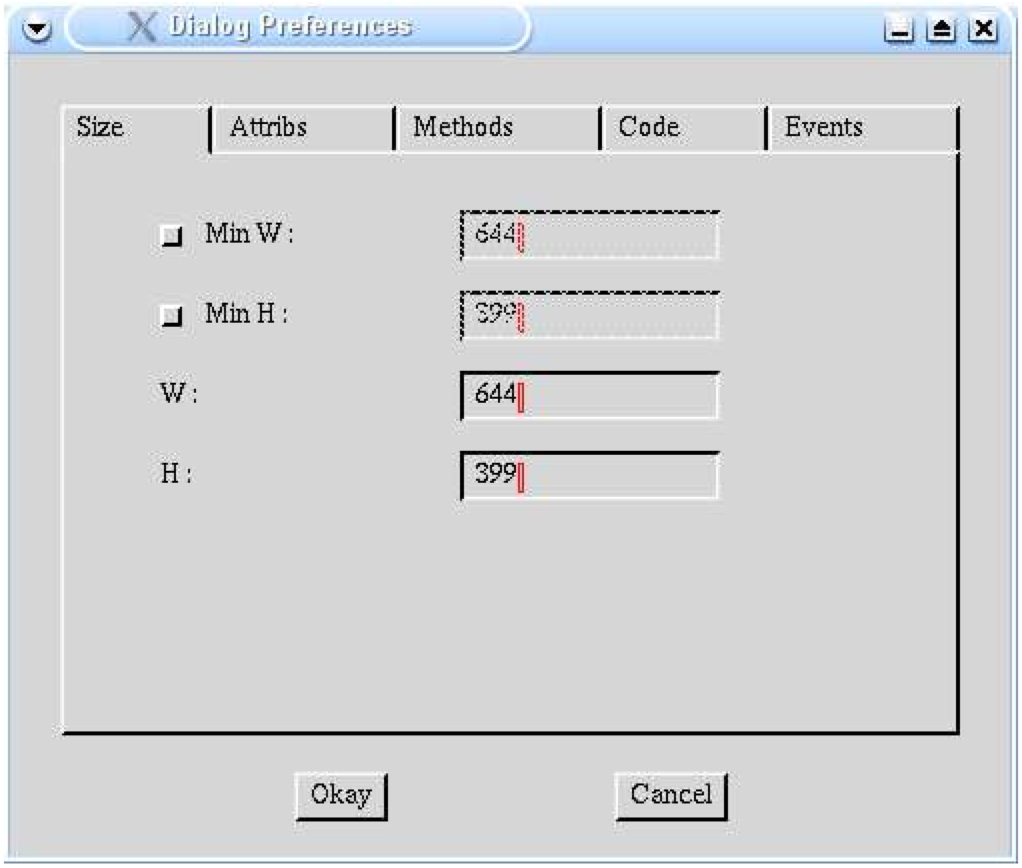
☐ Initially checked

Label alignment : c

Figure 12Button configuration window

Change the label by simply changing the text in the "Label" field and clicking "Okay".

As just mentioned, the dialog's attributes are initially the default window attributes. To change these, select the menu option Canvas -> Dialog prefs. The window shown below will appear.



Min W : 644
Min H : 399
W : 644
H : 399

Figure 13Dialog preferences window

To change the dialog attributes:

1. Click on the Attribs tab
2. Click on the Add button
3. Edit the two text fields so that they hold the attribute name and the attribute value respectively; for example try adding "bg" and "pale blue".
4. Click on Apply
5. Click on Okay.

Note that the button changes its background to pale blue too. Each object has its own attributes that it can set to override the dialog attributes. Click on the button and press Alt-D to bring up the button's configuration dialog again. Now click on the Attribs tab of this dialog and set the background color to white, for example. Then click okay and you will see that the button's background changes to white.

You will recall from the previous example programs that some objects can be contained in other objects, such as the `Panel` class. This is handled conveniently in Ivib. Add a `Panel` object to the dialog by clicking on the `Panel` button (on the second row, fourth from the left). A panel appears. Now drag the button into the panel. A message should appear in the information label below the toolbar, "Placed inside container." Now try dragging the panel about, and you will observe that the button moves too - it is now "inside" the panel. Dragging it outside the panel's area moves it back out. This method applies to all the container objects.

There are several buttons that operate on objects. The large "X" button deletes the currently selected objects. Try selecting the button and deleting it. The arrow buttons are "redo" and "undo" operations. Clicking on the undo button will undo the delete operation and the button should reappear.

Now try saving your canvas. Press Alt-S, and select a filename, or accept the default. At the end of an ivib-enhanced Unicon source file is a gigantic comment containing ivib's layout information. This comment is ASCII text, but it is not really human-readable. If the program is called, for example, `myprog.icn`, then this can be compiled with

```
    unicon myprog gui.u
```

to give an executable file `myprog` that, when run, will produce the same dialog shown in the canvas area. Of course, the resulting dialog will not do anything, and it is then up to the programmer to fill in the blanks by editing `myprog.icn`.

Hopefully, following the above steps will give you an idea of how the Ivib program works. Below are more details of how the individual components, dialogs, and operations work.

16.1 Moving, selecting and resizing

Select an object by clicking on it with the mouse. Its selection is indicated by red edges around the corners. Multi-select objects by holding the shift key down and clicking on the objects. The first selected object will have red corners; the others will have black corners. There are several functions which operate on multiple objects and map some attribute of the first selected object to the others; hence the distinction.

To move an object, select it by clicking on it with the left mouse button, and then drag it. Note that dragging an object even by one pixel will set the X or Y position to an absolute figure, disturbing any carefully set up percentage specification! Because this can be irritating when done accidentally, the X and/or Y position may be fixed in the dialog so that it cannot be moved in that plane. Alternatively, when selecting an object that you do not intend to move, use the right mouse button instead.

To resize an object, select it and then click on one of the corners. The mouse cursor will change to a resize cursor and the corner may be dragged. Note that, like the position specification, resizing an object will set the size to an absolute number and will also reset the "use default" option. Again, this can be avoided by fixing the width and/or height in the object's dialog box.

16.2 Dialog configuration

This dialog, accessed via the menu selection Canvas -> Dialog prefs allows the user to configure some general attributes of the dialog being created. The tabs are described in this section.

16.2.1 Size

The minimum width and height entries simply set the minimum dimensions of the window. The width and height may be configured here, or more simply by resizing the canvas area by clicking on the red bottom right-hand corner.

16.2.2 Attribs

This has been described briefly above; the Add button produces a new entry that is then edited. The edited entry is placed in the table with Apply. The Delete button deletes the currently highlighted selection.

16.2.3 Code generation

The part of the code to setup the dialog is written into a method called setup. If the "interpose in existing file" option is checked, then the program will read the present contents of the selected output file up to the current setup method, interpose the new setup, and copy the remainder out. This is useful if some changes have been made to the file output by a previous run. Note that it is important to take a copy of the existing file before using this option, in case unexpected results occur.

The other options simply select which other methods should be produced. If a main procedure is produced, then the result will be an executable program.

16.2.4 Other

This tab allows the name of the dialog to be set, together with a flag indicating whether it is "modal" or not. If so, then a method called pending is produced. This method is repeatedly called by the toolkit while it is waiting for events to occur.

16.3 Component configuration

Each component dialog has a standard tabbed pane area for configuration of attributes common to all components. The tabs are as follows:

16.3.1 Position and size

The X, Y, W and H options set the position and size of the object. The drop-down list can be used for convenience, or a value may be entered by hand. The "fix" buttons prevent the object from being moved or sized outside the given parameter in the Canvas area. This is useful once an object's position has been finalized, and you don't wish to accidentally move it. The "use default" buttons mean that the width/height will be set as the default for the object based on the parameters and the attributes. For example, a button's size will be based on the label and the font. For some objects there is no default size, so these buttons are shaded. The alignment of the object is also set from this tab.

16.3.2 Attribs

This works in exactly the same way as the Attribs tab for the dialog, except that these attributes apply only to the object.

16.3.3 Other

The Other tab allows the name of the object to be set. This is the name used in the output program code. The "Draw Border" button applies to some objects (see the reference section in Appendix C for further information). If the "Is Shaded" button is clicked, then the initial state of the object will be shaded. If the "Has initial focus" button is clicked, then this object will have the initial keyboard focus when the dialog is opened.

16.4 Component details

Components are added to the dialog by clicking on the toolbar buttons, and dialogs are produced by selecting the object and pressing Alt-D, as explained above. Most of the dialogs are hopefully straightforward, but some warrant further explanation.

16.4.1 TextButton

The dialog for this component includes an option for the button to be added to a ButtonGroup structure. This is explained in detail shortly.

16.4.2 Border

The Border component is rather odd. To select it, rather than clicking inside the border, click in the area at the bottom right hand corner. It can then be resized or moved. Now try dragging another object, such as a CheckBox or Label and release it so that its top left-hand corner is within the area in the bottom right-hand corner of the Border object. The CheckBox/Label or whatever is now the title of the Border. Thus, any object can be in the title. To remove the object from the Border, just drag it out. The alignment of the title object is set in the dialog, but is by default left aligned.

16.4.3 Image

Initially the Image object has an outline. When a filename is entered into the dialog however, the image itself is displayed.

16.4.4 Checkbox

Customized up/down images may be set from the dialog, and a `CheckBoxGroup` may be selected if one is available; this is explained in more detail shortly.

16.4.5 MenuBar

The `MenuBar` dialog enables a complete multi-level menu to be created. Clicking on a particular line will allow an item to be inserted at that point. (Only menus can be inserted into the lowest level of course). Clicking on an item will allow insertion, deletion, or editing of the particular item. A `CheckBoxGroup` can be created by selecting multiple check boxes (by holding down the Shift key while clicking on the lines) and clicking the button.

16.4.6 ScrollBar

Both vertical and horizontal scroll bars are available; for details of how the various options work, please see the reference manual for the toolkit.

16.4.7 Table

A table column is added by clicking the Add button; the details should then be edited and the Apply button pressed to transfer the details to the table. A selected column can be deleted with the Delete button. The drop-down list selects whether or not lines in the table can be selected by clicking them.

16.4.8 TabSet

Add a `TabItem` (a tabbed pane) by clicking the Add button. Note that a single pane is automatically present when the object is created. To switch between panes, select a `TabItem` button. An asterisk appears by the entry, and when the dialog is exited, it is this `TabItem` that is to the front of the `TabSet`. To add items to the current pane, simply drag and drop them into it. The whole of the item must be in the pane, and a confirmatory message appears to indicate that the item has been added to the container. To take it out of the container, just drag it out of the pane. Note that the selected pane is the one that is configured to be initially at the front when the dialog is opened.

16.4.9 MenuButton

The MenuButton component is just a menu system with one root menu. The dialog is the same as that for `MenuBar` except that the small icon can be configured.

16.4.10 OverlaySet

The configuration for an `OverlaySet` is very similar to that for a `TabSet`, except that there are no tab labels to configure of course.

16.4.11 CheckBoxGroup

This does not create an object on the screen, but rather places several selected `CheckBox` objects into a `CheckBoxGroup` object, so that they act as coordinated radio buttons. To use this button, select several `CheckBox` objects, and press the button.

The `CheckBoxGroup` itself is configured by selecting the menu item Canvas -> CheckBoxes. In fact, the only attribute to be configured is the name of the `CheckBoxGroup`. Note that once a `CheckBoxGroup` has been created, it cannot be deleted. A `CheckBox` can be taken out or put into a `CheckBoxGroup` from its configuration dialog.

16.4.12 ButtonGroup

The ButtonGroup operates in a very similar fashion to CheckBoxGroup, except that it places buttons into a ButtonGroup.

16.5 Other editing functions

16.5.1 Delete

The Delete function simply deletes all the selected objects; note that deleting a container also deletes all the objects inside it.

16.5.2 Undo and Redo

The Undo and Redo functions undo and redo changes. The size of the buffer used for storing undo information can be configured in the File -> Preferences dialog; by default it allows 7 steps backward at any one time.

16.5.3 Center Horizontally

The Center Horizontally operation sets the selected objects' X position specification to "50%", their alignment to center, and fixes them in that horizontal position. To "unfix" an object, uncheck the "fix" box in its dialog box. Center vertically naturally works in just the same way for the y position.

16.5.4 Align Horizontally

The Align horizontally operation sets the X position and the X alignment of all the selected objects to the X position and the X alignment of the first selected object. Note that whether the objects end up appearing to be left aligned, center aligned, or right aligned will depend on the alignment of the first selected object. "Align vertically" works just the same way.

16.5.5 Grid

To use the Grid function, place several items roughly in a grid, select them all, perform the operation, and hopefully they will be nicely aligned.

16.5.6 Copy

The Copy function simply creates a duplicate of each selected object.

16.5.7 Equalize widths

The Equalize Widths function simply copies the width of the first selected object to all of the other selections. "Equalize heights" naturally does the same for heights.

16.5.8 Even Space Horizontally

The Even Space Horizontally operation leaves the leftmost and rightmost object of the current selections in place and moves all of the other objects so that they are equally spaced between the leftmost and rightmost objects. "Even Space Vertically" does the same vertically.

16.5.9 Reorder

The Reorder function is used to reorder the selected objects in the program's internal lists. This is useful so that they are produced in the program output in the desired order, for example to ensure that the tab key moves from object to object in the right sequence. By selecting several objects and using reorder, those objects appear first in sequence in the order in which they were selected.

17 Summary

The Unicon GUI toolkit offers a full-featured and attractive way of constructing interfaces. The toolkit has modern features - such as tables, tabbed property sheets, and multiline text editing capability - that are not present in Icon's earlier vidgets library. Many components that are present in both libraries are more flexible in the GUI toolkit, supporting fonts, colors, and graphics that the vidgets library does not handle well. The ivib interface builder tool provides programmers with easy access to the GUI toolkit.

The object-orientation of the class library mainly affects its extensibility, although arguably it may also contribute to the simplicity of the design. Inheritance, including multiple inheritance, is used extensively in the 37 classes of the GUI toolkit. Inheritance is the main object-oriented feature that could not be easily mimicked in a procedural toolkit such as the vidgets library, and inheritance is the primary extension mechanism.