Using STklos for graphic programming

or

"Programming the Tk toolit with an OO Scheme"

Erick Gallesio (`eg@unice.fr`)
any other volunteers are welcome

April 1995

Contents

Chapter 1

Getting Started

This chapter will explain the things you must know to begin to manipulate Tk widgets as STklos objects. This is a short introduction for programming the Tk toolkit with objects. If you know how to program it with the Tcl language, you will see that things are not too much different (at least at first sight). A good introduction to Tk programming in Tcl can be found in [?] or [?].

1.1 First steps

When you want to use the Tk toolkit with the STklos, you must first call the STk interpreter. Launching the interpreter is usually done by a call to the shell script **stk** which must have been installed in a standard place. Once, the interpreter is initialized, a small square window will appear on your display. This window is called the *root window*. Its value is always retained in the global variable `*top-root*`.

Once the Tk initialization is complete, you have to load the file `Tk-classes`. This can be done by

```
(require "Tk-classes")
```

The `Tk-classes` file contains a set of `autoloads` for all the Tk widgets defined in the STk release. If you plan to always work with those classes, you can put this line in your STk init file (i.e. in the file "`~/.stkrc`") to avoid (manual) loading of this file.

From now, we are able to interact with the Tk toolkit. Try to enter the following lines at the STk prompt:

```
(define hello (make <Label> :text "Hello, world"))
```

The call to **make** creates a new object (i.e. an instance of the class **<Label>**) with its slot **text** filled with the string `"Hello, world"`. In the above example, this new object is assigned to the symbol `hello`. Even, if a new label is created, nothing will be displayed on your screen. You'll have to use a *geometry manager* for displaying a Tk widget onto screen. Two window managers are provided with Tk, namely the *packer* and the *placer*. Both command have a full set of options which will be detailed later (see ??). For now, we will just use the packer in its simpler form:

```
(pack hello)
```

permits to map the `hello` button onto screen.

Let us see further how the `hello` button is built. Using describe on it will display:

```
#[<label> e1444] is an instance of class <label>
Slots are:
     id = #[Tk-command .v1]
     eid = #[Tk-command .v1]
     parent = #[Tk-command .]
     bitmap = ""
     width = 0
     height = 0
     anchor = "center"
     font = "9x15"
     foreground = "Black"
     pad-x = 1
     pad-y = 1
     text = "Hello, world"
     text-variable = ""
     background = "#cccccc"
     border-width = 0
     cursor = ""
     relief = "flat"
```

`Id`, `Eid` and `parent` slots are present in all Tk widgets and are used by the system. Don't try to change their value. Other slots depend of the kind (i.e. the class) of widget. The slots listed above correspond to the various Tk options available on a label. Of course you can consult/change their value with the standard `slot-ref` or `slot-set!` primitives. For instance, one can change the background color of the `hello` button with the following form:

```
(slot-set! hello 'background "Chocolate")
```

or, using the STklos generalized assignment,

```
(set! (background hello) "Chocolate")
```

Of course, the background value can be queried with

```
(slot-ref hello 'background)
       --> "Chocolate"
```

or

```
(background hello)
       --> "Chocolate"
```

Here, `background` is used as an accessor of the label background. It would be also possible to specify this value at label instantiation with the `:background` *initialization keyword* and the same window could have be obtained with the following `make` call

```
(define hello (make <Label> :text "Hello world"
                            :background "Chocolate"))
```

1.2 Class Hierarchy

STklos Tk classes permit to *reify* the Tk widgets into STklosobjects. It means that every graphical object used in a program such as a menu, a label or a button is represented as a

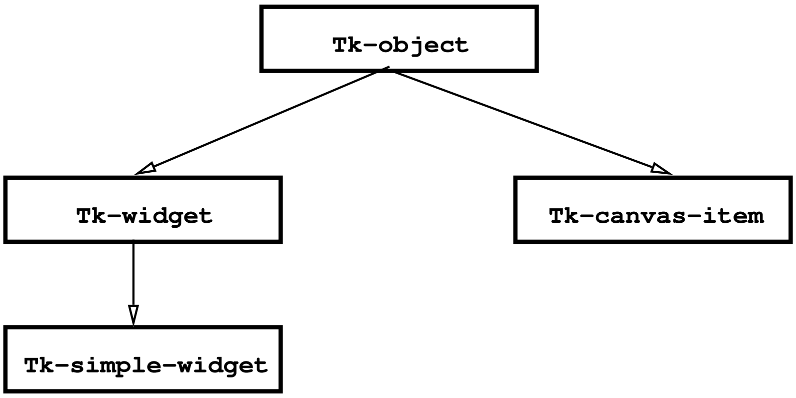

Figure 1.1: Head of the STkLOS hierarchy

STkLOS object. All the defined STkLOS classes build a hierarchy which is briefly described here. Firstly, all the classes shared a unique ancestor: the `<Tk-object>` class. Instances of this class contain informations which are necessary for establishing a communication between the Scheme and Tk worlds. Principally, objects of this class have the three slots cited before `Id`, `Eid` and `parent`. The `Id` slot contains a Tk command [?], generated by STkLOS itself, which correspond to the STk command which implement the object. The `parent` slot contains a reference to the object which (graphically) include the current object. This slot will be discussed later. Normally, end users will not have to use direct instances of the `<Tk-object>` class[1].

The next level in our class hierarchy define a fork with two branches: the `<Tk-widget>` class and `<Tk-canvas-item>` class. Instances of the former class are classical widgets such as buttons, menus or canvases since instances of the later are objects contained in a canvas such as lines or rectangles. `<Tk-widget>`s are also divided in two categories: `<Tk-simple-widget>`s and `<Tk-composite-widget>`s. Simples widgets are directly implemented as Tk objects and composite ones are build upon simple widgets (e.g. file browser, alert messages and so on). The head of the STkLOS hierarchy is shown in Figure 1.1.

1.2.1 The `<Tk-widget>` class

No slot is defined for this class. However, since this class is common to all the interface objects (menus, buttons, label, ...), it permits to define their common behavior. The behavior of those objects is described using STkLOS methods with one of their argument (generally the first) which is a `<Tk-widget>`. The *pack* geometry manager cited below is one of such methods. The *packer* will be fully described below (see 9); however, briefly stated, we can say that it permits to explain the geometry of a widget depending of its neighbors. The example below shows how to use the pack primitive with arguments.

```
(define lab1 (make <Label> :text "Label 1"))
(define lab2 (make <Label> :text "Label 2"))

(pack lab1 :side "bottom")
(pack lab2 :side "top" :expand #t :fill "both")
```

Here, two labels are defined `lab1` and `lab2`. The first call to pack says that `lab1` should occupy the bottom of the window. Second call states that `lab2` will occupy the top of the window.

[1] All classes whose name begin with the "Tk-" prefix are not are not intended for the final user. They will be discussed in this document only if it seems necessary for a good class hierarchy comprehension.

Furthermore, the options `:expand` and `:fill` specified during `lab2` packing tells to the packer that its geometry must be adjusted to fill all the space if its containing window is resized. If we do now

```
(define lab3 (make <Label> :text "Label 3"))
(pack lab3 :after lab2)
```

we will define a new label which will be inserted between the `lab1` and `lab2` labels. Of course, the window containing the two labels will be *automagically* resized to take into account the insertion of the third.

Windows can also be unmapped from screen with the `unpack` method. This method tells *pack* to suspend geometry management of the specified window. Another call to a geometry manager should be used to map again the window on display.

Suppose now that we want to delete the top most label of our previous window (i.e. `lab2`). This can be done with the `<Tk-widget>` method `destroy` as shown below:

```
(destroy lab2)
```

Here, the window is unmapped from screen and definitively destroyed (i.e. it cannot be mapped on the screen anymore). Technically speaking, the widget will be destroyed and its class will be changed to the special class `<Destroyed-object>`.

Note: Destroying the root window (`*top-root*`) will destroy the root window and, consequently, the interpreter since it as nothing more to manage.

1.2.2 The `<Tk-simple-widget>` class

STklos defines a class for each simple widgets (such as labels, buttons or messages) of the Tk library. The `<Tk-simple-widget>` class defines slots and methods which are common to all those objects. The main slots defined for simple widgets are:

background

which specifies the normal background color to use when displaying the widget. The value of this slot should be a *string* or a *symbol*. It can be

- a textual name for a color defined in the server's color database file (e.g. "Chocolate" or "Dark Olive Green").
- a numeric specification of the red, green, and blue intensities to use to display the color (such as `#aabbcc` or even `rgb:aa/bb/cc`) using the standard X11 notation.

border-width

which specifies a non-negative value indicating the width of the 3-D border to draw around the outside of the widget (if such a border is being drawn; the *relief* option typically determines this).

cursor

which specifies the mouse cursor to be used for the widget. Valid values for this slot are given in Table 1.1. Be aware that case is significant when naming a cursor.

relief

X_cursor	fleur	sailboat
arrow	gobbler	sb_down_arrow
based_arrow_down	gumby	sb_h_double_arrow
based_arrow_up	hand1	sb_left_arrow
boat	hand2	sb_right_arrow
bogosity	heart	sb_up_arrow
bottom_left_corner	icon	sb_v_double_arrow
bottom_right_corner	iron_cross	shuttle
bottom_side	left_ptr	sizing
bottom_tee	left_side	spider
box_spiral	left_tee	spraycan
center_ptr	leftbutton	star
circle	ll_angle	target
clock	lr_angle	tcross
coffee_mug	man	top_left_arrow
cross	middlebutton	top_left_corner
cross_reverse	mouse	top_right_corner
crosshair	pencil	top_side
diamond_cross	pirate	top_tee
dot	plus	trek
dotbox	question_arrow	ul_angle
double_arrow	right_ptr	umbrella
draft_large	right_side	ur_angle
draft_small	right_tee	watch
draped_box	rightbutton	xterm
exchange	rtl_logo	

Table 1.1: Valid cursor values for slot `cursor`

which specifies the 3-D effect desired for the widget. Acceptable values are `:raised`, `:sunken`, `:flat`, `:ridge` or `:groove`. The value indicates how the interior of the widget should appear relative to its exterior; for example, `:raised` means the interior of the widget should appear to protrude from the screen, relative to the exterior of the widget.

These slots are defined for all the basic widgets of the STklos package. However, they are not defined as real STklos slots (i.e. they are not implanted in the Scheme world). Instead, this kind of slot has its value which is stored inside the internal structures of the Tk library. Those slots are allocated with a special allocation scheme which is called `:tk-virtual`. Tk virtual slots are managed by the `<With-Tk-virtual-slots-metaclass>` meta-class.

Consequently, reading or writing a *tk-virtual* slot will provoke a direct reading or writing of a field of the Tk-library C-structure associated to the object (this avoid to have to synchronize values in the Tk and Scheme world). For each *tk-virtual*, STklos provides an accessor; the following example shows some simple usages of this kind of slots.

Example:

```
(define lab (make <Label> :text "Hello" :relief "raised" :background "grey"))
(relief lab)
        --> "raised"
(set! (border-width lab) 4)
(border-width lab)
        --> 4
```

The first expression creates a new instance of a label, as we have seen before. The second one queries the Tk server about the relief of the label `lab`. Finally, the `set!` expression permits to change the value of the width of `lab`'s border.

1.2.3 The `<Tk-canvas-item>` class

Instances of `<Tk-canvas-item>` class, as said before, are objects such as rectangles, circles or bitmaps which are contained in a window. A window containing this kind of object is a special simple widget called a *canvas*. All the objects defined in a *canvas* can be manipulated (e.g. moved, re-colored or re-sized) and even Scheme command can be associated to them. All the actions available for `<Tk-canvas-item>`s are deferred to a next chapter (??).

1.3 Functions of this chapter

This section presents in details the methods and functions seen in this chapter (or methods and functions related to things seen in this chapter). Functions are listed below in alphabetical order.

`destroy` `method`

Syntax

```
destroy ((self <Tk-widget>))
destroy l
```

Arguments (first form)

self the window to destroy.

Arguments (second form)

l a list of window to destroy (this form maps in facts the previous one to all its arguments to allow multiple destruction).

Description

`Destroy` deletes the `self` widget window and all of its descendants. If `self` equals to `*top-root*`, the interpreter terminates its execution.

Result

The destroyed object (see note below)

Example

```
(define l1 (make <Label> :text "lab1"))
...
(define l5 (make <Label> :text "lab5"))
(destroy l1)            ;; direct call of first form
(destroy l2 l3 l4 l5)   ;; call of the second form
```

Note

When a widget is destroyed, its class is changed to ¡Destroyed-object¿ as shown below:

```
(define lab (make <Label> :text "A label"))
(class-name (class-of lab))
        --> <label>
(destroy lab)
(class-name (class-of lab))
        --> <destroyed-object>
```

See also

`destroy` method for canvases, unpack

`pack` — procedure

Syntax

`pack l`

Arguments

`pack` accepts a list of arguments whose behavior depends on the first element of this list.

`(pack 'slave [slave ...] [option])`

If the first argument to `pack` is a widget, then the command is processed in the same way as `(pack 'configure ...)`.

`(pack 'configure [slave ...] [option])`

The arguments consist of one or more slave windows followed by pairs of arguments that specify how to manage the slaves. See "THE PACKER ALGORITHM" below for details on how the options are used by the packer. The following options are supported:

`:after other`
`:before other`
: `Other` must the name of another window. Use its master as the master for the slaves, and insert the slaves just after (or before) `other` in the packing order.

`:anchor anchor`
: `Anchor` must be a valid anchor position such as "`n`" or "`sw`"; it specifies where to position each slave in its parcel. Defaults to "`center`".

`:expand boolean`
: Specifies whether the slaves should be expanded to consume extra space in their master. Defaults to `#f`.

`:fill style`
: If a slave's parcel is larger than its requested dimensions, this option may be used to stretch the slave. `Style` must have one of the following values:

  "none" Give the slave its requested dimensions plus any internal padding requested with `:ipadx` or `:ipady`. This is the default.

  "x" Stretch the slave horizontally to fill the entire width of its parcel (except leave external padding as specified by `:padx`).

  "y" Stretch the slave vertically to fill the entire height of its parcel (except leave external padding as specified by `:pady`).

  "both" Stretch the slave both horizontally and vertically.

`:in other`
: Insert the slave(s) at the end of the packing order for the master window given by other.

`:ipadx amount`
`:ipady amount`

`Amount` specifies how much horizontal (or vertical) internal padding to leave on each side of the slave(s). `Amount` must be a valid screen distance, such as 2 or ".5c". It defaults to 0.

`:padx amount`
`:pady amount`

`Amount` specifies how much horizontal external padding to leave on each side of the slave(s). `Amount` defaults to 0.

`:side side`

Specifies which side of the master the slave(s) will be packed against. Must be "`left`", "`right`", "`top`", or "`bottom`". Defaults to "`top`".

If no `:in`, `:after` or `:before` option is specified then each of the slaves will be inserted at the end of the packing list for its parent unless it is already managed by the packer (in which case it will be left where it is). If one of these options is specified then all the slaves will be inserted at the specified point. If any of the slaves are already managed by the geometry manager then any unspecified options for them retain their previous values rather than receiving default values.

`(pack 'newinfo slave)`

Returns a list whose elements are the current configuration state of the slave given by slave in the same option-value form that might be specified to `pack 'configure`. The first two elements of the list are "`-in master`" where master is the slave's `master`. Starting with Tk 4.0 this option will be renamed "`(pack 'info ...)`".

`(pack 'propagate master [boolean])`

If `boolean` is `#t` then propagation is enabled for `master`, which must be a window name (see "GEOMETRY PROPAGATION" below). If `boolean` has a false boolean value then propagation is disabled for `master`. In either of these cases an empty string is returned. If `boolean` is omitted then the command returns 0 or 1 to indicate whether propagation is currently enabled for `master`. Propagation is enabled by default.

`(pack 'slaves master)`

Returns a list of all of the slaves in the packing order for `master`. The order of the slaves in the list is the same as their order in the packing order. If `master` has no slaves then an empty string is returned.

Description

THE PACKER ALGORITHM For each master the packer maintains an ordered list of slaves called the *packing list*. The `:in`, `:after`, and `:before` configuration options are used to specify the master for each slave and the slave's position in the packing list. If none of these options is given for a slave then the slave is added to the end of the packing list for its parent.

The packer arranges the slaves for a master by scanning the packing list in order. At the time it processes each slave, a rectangular area within the master is still unallocated. This area is called the *cavity*; for the first slave it is the entire area of the master.

For each slave the packer carries out the following steps:

The packer allocates a rectangular *parcel* for the slave along the side of the cavity given by the slave's `:side` option. If the side is top or bottom then the width of the parcel is the width of the cavity and its height is the requested height of the slave plus the `:ipady` and `:pady` options. For the left or right side the height of the parcel is the height of the cavity and the width is the requested width of the slave plus the `:ipadx` and `:padx` options. The parcel may be enlarged further because of the `:expand` option (see "EXPANSION" below)

2. The packer chooses the dimensions of the slave. The width will normally be the slave's requested width plus twice its `:ipadx` option and the height will normally be the slave's requested height plus twice its `:ipady` option. However, if the `:fill` option is "`x`" or "`both`" then the width of the slave is expanded to fill the width of the parcel, minus twice the `:padx` option. If the `:fill` option is "`y`" or "`both`" then the height of the slave is expanded to fill the width of the parcel, minus twice the `:pady` option.

3. The packer positions the slave over its parcel. If the slave is smaller than the parcel then the `:anchor` option determines where in the parcel the slave will be placed. If `:padx` or `:pady` is non-zero, then the given amount of external padding will always be left between the slave and the edges of the parcel.

Once a given slave has been packed, the area of its parcel is subtracted from the cavity, leaving a smaller rectangular cavity for the next slave. If a slave doesn't use all of its parcel, the unused space in the parcel will not be used by subsequent slaves. If the cavity should become too small to meet the needs of a slave then the slave will be given whatever space is left in the cavity. If the cavity shrinks to zero size, then all remaining slaves on the packing list will be unmapped from the screen until the master window becomes large enough to hold them again.

EXPANSION If a master window is so large that there will be extra space left over after all of its slaves have been packed, then the extra space is distributed uniformly among all of the slaves for which the `:expand` option is set. Extra horizontal space is distributed among the expandable slaves whose `:side` is "`left`" or "`right`", and extra vertical space is distributed among the expandable slaves whose `:side` is "`top`" or "`bottom`".

GEOMETRY PROPAGATION The packer normally computes how large a master must be to just exactly meet the needs of its slaves, and it sets the requested width and height of the master to these dimensions. This causes geometry information to propagate up through a window hierarchy to a top-level window so that the entire sub-tree sizes itself to fit the needs of the leaf windows. However, the `pack 'propagate` command may be used to turn off propagation for one or more masters. If propagation is disabled then the packer will not set the requested width and height of the packer. This may be useful if, for example, you wish for a master window to have a fixed size that you specify.

RESTRICTIONS ON MASTER WINDOWS The master for each slave must either be the slave's parent (the default) or a descendant of the slave's parent. This restriction is necessary to guarantee that the slave can be placed over any part of its master that is visible without danger of the slave being clipped by its parent.

PACKING ORDER If the master for a slave is not its parent then you must make sure that the slave is higher in the stacking order than the master. Otherwise the master will obscure the slave and it will appear as if the slave hasn't been packed correctly. The easiest way to make sure the slave is higher than the master is to create the master window first: the most recently created window will be highest in the stacking order. Or, you can use the `raise` and `lower` commands to change the stacking order of either the master or the slave.

Result

None

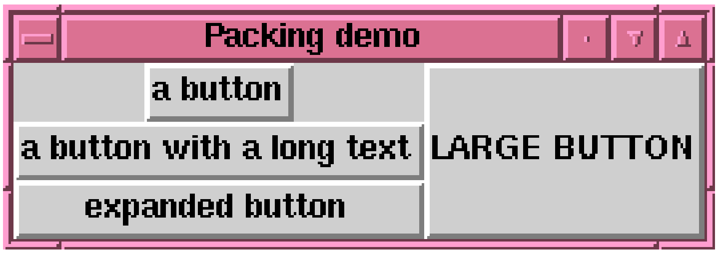

Figure 1.2: Using the pack options

Example

```
(define b1 (make <Button> :text "a button"))
(define b2 (make <Button> :text "a button with a long text"))
(define b3 (make <Button> :text "expanded button"))
(define b4 (make <Button> :text "LARGE BUTTON"))
(pack b1 b2)
(pack b3 :expand #t :fill "x")
(pack b4 :expand #t :fill "both" :side "right" :before b1)
```

These calls to `pack` will produce the output shown in Figure 1.2.

See also

`place`, `unpack`

`place` — procedure

Syntax

`place l`

SEE TK MAN PAGE :-¡

`unpack` — procedure

Syntax

`unpack  l`

Arguments

l a list of widgets to unmap.

Description

`Unpack` is a simple way to unmap the widgets of the `l` list. This procedure calls in fact `pack 'forget` procedure. After the execution of this procedure the *pack* geometry manager will no longer manage the geometry of given widgets.

Result

Undefined

Example

(define l1 (make ¡Label¿ :text ”lab1”)) ... (define l5 (make ¡Label¿ :text ”lab5”)) (destroy l1) ;; Object is unmapped AND destroyed (unpack l2 l3 l4 l5) ;; Objects are ONLY unmapped (class-name (class-of l1)) −¿ ¡destroyed-object¿ (class-name (class-of l2)) −¿ ¡label¿

See also

`place`, `pack`, `destroy`

Chapter 2

Basic widgets

This chapter is devoted to the simplest widgets of the STklos package. In fact, all the Tk widgets, except the *canvas* and *text* widgets, are presented here.

2.1 Introduction

2.1.1 A simple interface

Defining the widgets

Before detailing all the simple widgets of the Tk library, we will see how to build the simple interface which is shown in Figure 2.1. This interface is composed of three main components:

1. a label which is situated at top
2. a frame which is itself constituted of two components:
   (a) a listbox, and
   (b) a scrollbar
3. a "quit" button

Let's have look now to the objects we'll have to create for this interface. First we can create the top label and the quit button which are relatively simple. This can be done with the following definitions:

```
(define lab  (make <Label>  :text "A simple interface"))
(define quit (make <Button> :text "Quit" :command '(exit)))
```

The `:command` init-keyword permits to associate an action to the button (this action will be triggered when the mouse button 1 will be released over the `quit` button.

The interface shown in Figure 2.1 can be seen as a vertical alignment of three components (a label, a listbox with a scrollbar and a button). The second component is itself an alignment of two object (a scrollbar and a listbox). What is important to note here is that this alignment is an horizontal one situated inside a vertical one. In this case, we have to place the components of our horizontal alignment in a container which acts as a kind of back box for further layout. The widget which permit to "embody" several widgets is called a frame in the Tk toolkit. Consequently, the listbox and its scrollbar can be defined by:

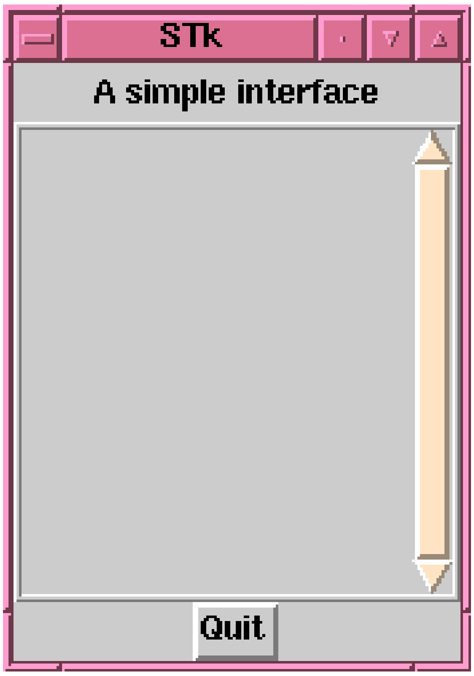

Figure 2.1: A simple interface

```
(define box  (make <Frame> :border-width 3 :relief "ridge"))
(define l    (make <Listbox>   :parent box))
(define s    (make <Scrollbar> :parent box :orientation "vertical"))
```

What is important to note is that the listbox and the scrollbar both tell that their *parent* is the frame `box`. This is this indication which effectively embody them in the frame.

All the basic components of our interface have been created. What it rests to do consists to place the graphic components we have just defined onto screen. First we can map the listbox and its scrollbar in the `box` object with the *packer* geometry manager (see page 10):

```
(pack l :expand #t :fill "both" :side "left")
(pack s :expand #t :fill "y" :side "right")
```

Both calls to `pack` permit to define the arrangement of the listbox and the scrollbar into the frame. Here we claim that the scrollbar should stay on the right side of the frame and that it must expand itself only in the vertical direction if its containing frame (i.e. `box`) is resized. Concerning the listbox, it will stay on rigth and it will fill all the space (`:fill "both"`) if `box` size changes.

Now we can *pack* the three components of this simple interface to show up them on screen; it can be easily done by:

```
(pack lab box quit)
```

The complete program for the building the previous interface is given in Figure 2.2.

Filling the listbox

Filling the listbox can be done by calling the `insert` method. This method permits to add elements in the listbox. A way to fill the previous lisbox could be:

```
;;; Defining the components
(define lab  (make <Label>  :text "A simple interface"))
(define quit (make <Button> :text "Quit" :command '(exit)))
(define box  (make <Frame>  :border-width 3 :relief "ridge"))

;;; Zoom in the box
(define l    (make <Listbox>   :parent box))
(define s    (make <Scrollbar> :parent box :orientation "vertical"))

(pack l :expand #t :fill "both" :side "left")
(pack s :expand #t :fill "y" :side "right")

;;; Pack the components
(pack lab box quit)
```

Figure 2.2: Complete code for the simple interface

```
(insert l 0 "a" "b" "c")
```

This will insert 3 lines (containing `"a"`, `"b"` and `"c"`) after the element whose index is 0. Inserting a `"d"` between `"a"` and `"b"` could be done by the following expression:

```
(insert l 1 "d")
```

Deleting elements of listbox necessitates a call to the `delete` method. For instance,

```
(delete l 1)
```

deletes the second element of the listbox (first element is at index 0), and

```
(delete l 0 2)
```

permits to delete the three remaining elements.

`Insert` and `delete` are simple methods built upon Tk commands to accesss listbox. Most of the time, this way of accessing a listbox is not too convenient, and it is more easy to see the content of a listbox as its value. Consequently, the `stklos <Listbox>` class defines a virtual slot[?] called `value` which permits to read/fill the contents of a listbox as a STklos slot. For instance, the previous listbox can be filled with:

```
(set! (value l) '("a" "b" "c"))  ;; or (slot-set! l 'value '(...))
```

and the value of the second element of the listbox can be obtained by

```
(list-ref (value l) 1)
```

Of course, the value slot has an initialization keword associated to it, and the filling of the listbox could have been done during the definition of the listbox:

```
(define l (make <Listbox> :parent box
                          :value  '("a" "b" "c")))
```

Bringing the scrollbar to life

For now, the lisbox and the scrollbar are disconnected (i.e. moving the listbox, with `<Shift-Button2>` doesn't move the scrollbar and clicking the scrollbar does nothing). To make both widgets working accordingly, we have to associate a command to each widget. First, we can associate a command to the scrollbar, such as moving it with mouse button will move the listbox. This can be done with:

```
(set! (command s) (format #f "y-view ~S " (address-of l)))
```

which associates the prefix of a Scheme command to invoke to change the view in the listbox when the user manipulates the scrollbar. This command is partial and will be completed effectively when the user will click the scrollbar (see the ¡Scrollbar¿ class description on page ?? for more details).

This is at half way of what we have to do for cleanly associate the scrollbar and the listbox. On the listbox side, we have to associate a command for its vertical movement:

```
(set! (y-scroll-command l)
      (format #f "scrollbar-set! ~S " (address-of s)))
```

Here again, the command associated to the listbox is a partial since it will be completed by Tk when needed.

As we will see in a next chapter, *composite widgets* will permit to hide all this stuff, and we will be able to define a new class which will embody a listbox and one (or two) scrollbar.

2.1.2 Defining Menus

Menu-buttons and Menus

Tk provides two widgets for building a menu: *menubuttons* and *menus*. Those widgets are available through the `<Menu-button>` and `<Menu>` classes. Instances of `<Menu-button>` class display a textual string (or a bitmap) and are associated with an instance of the `<Menu>` class.

Creating a simple menu bar

The most current task with menus concerns menu bars building. STklos provides the convenience function `make-menubar` to ease to the construction of menu bars. To illustrate how this function work, we'll present the call we must write to produce the menu bar of Figure 2.3. `Make-menubar` is a function which takes a list of menu buttons specifications. Each menu button specification is also a list. First item of this list is the specification of the menu button; the rest of this list correspond to the specifaction of the menu items associated to this menu button. To represent the menu bar of Figure2.3, we have a specification whih looks like:

```
((Description of  Work days menu button
     (Fisrt item of  Work days)
     (Second item of  Work days))
 (Description of  Week end menu button
     (Fisrt item of  Week-end)
     (Second item of  Week-end))
)
```

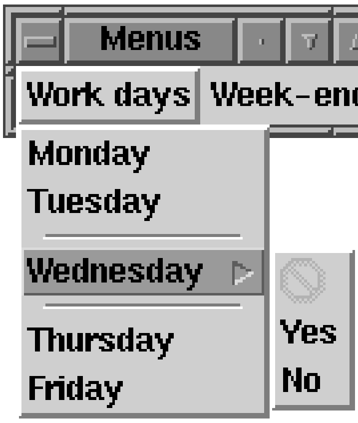

Figure 2.3: A simple menu

```
(define l ‘(("Work days"
              ("Monday"    ,(lambda () (format #t "Monday\n")))
              ("Tuesday"   ,(lambda () (format #t "Tuesday\n")))
              ("")
              ("Wednesday"
                  ((command :bitmap error :state disabled)
                   ("Yes" ,(lambda () (format #t "Wednesday (Yes)\n")))
                   ("No"  ,(lambda () (format #t "Wednesday (No) \n")))))
              ("")
              ("Thursday"  ,(lambda () (format #t "Thursday\n")))
              ("Friday"    ,(lambda () (format #t "Friday\n"))))
        ("Week-end"
              ("Saturday"  ,(lambda () (format #t "Saturday\n")))
              ("Sunday"    ,(lambda () (format #t "Sunday\n"))))))

(pack (make-menubar *top-root* l))
```

Figure 2.4: Code of menu of Figure2.3

The complete code for this menu bar is given in Figure2.4. Struture of menu buttons and menu items specifications is defined in ??.

2.2 Simple widgets classes

2.2.1 The `Frame` class

A frame is a STklos simple widget whose primary purpose is to act as a spacer or container for complex window layouts. A frame object has several Tk-virtual slots which are listed here:

background

See section 1.2.2.

border-width

See section 1.2.2.

cursor

See section 1.2.2.

height

specifies the desired height for the window. This slot is only used if the *geometry* slots is unspecified. If this pseudo-slot is less than or equal to zero (and *geometry* is not specified) then the window will not request any size at all.

geometry

specifies the desired geometry for the widget's window, in the form *width*x*height*, where *width* is the desired width of the window and *height* is the desired height. The units for *width* and *height* depend on the particular widget. For widgets displaying text the units are usually the size of the characters in the font being displayed; for other widgets the units are usually pixels.

relief

See section 1.2.2.

width

specifies the desired width for the window. This slot is only used if the *geometry* slots is unspecified. If this pseudo-slot is less than or equal to zero (and *geometry* is not specified) then the window will not request any size at all.

Furthermore, the *Frame* class admits a special initarg `:class` which permits to specify the new widget X11 resource class (if the `:class` is not passed as a parameter during the `make-instance` of the frame, the widget resource class will be "Frame"). The resource class of a widget can be specified only at initialization. It can be queried by the `class` reader but cannot be changed.

Further example use a frame to group three labels in a single object. Here, the two

```
(setq f (make-instance 'Frame))
(setq l1 (make-instance 'Label :parent f :text " Label 1 " :relief :raised
                                :map '(:position :left)))
(setq l2 (make-instance 'Label :parent f :text " Label 2 " :relief :raised
                                :map '(:position :left)))
(setq l3 (make-instance 'Label :text " Label 3 " :relief :raised))
```

2.2.2 The label class

The *Label* class defines widgets which are capable to display a textual string or a bitmap. A label object has several pseudo-slots which are listed here:

anchor

specifies how the information in a widget (e.g. text or a bitmap) is to be displayed in the widget. Must be one of the values *:n*, *:ne*, *:e*, *:se*, *:s*, *:sw*, *:w*, *:nw*, or '*:center*. For example, *:nw* means display the information such that its top-left corner is at the top-left corner of the widget.

background

See section 1.2.2.

bitmap

specifies a string containing the file name of the bitmap to display in the widget. The exact way in which the bitmap is displayed may be affected by other slots values such as *anchor* or *justify*. Typically, if this slot is non `nil` is specified then it overrides other pseudo-slots that specify a textual value to display in the widget; the *bitmap* slot may be reset to an empty string to re-enable a text display.

border-width

See section 1.2.2.

cursor

See section 1.2.2.

font

specifies the font to use when drawing text inside the widget.

foreground

specifies the normal foreground color to use when displaying the widget.

height

specifies a desired height for the label. If a bitmap is being displayed in the label then the value is in screen units; for text it is in lines of text. If this pseudo-slot isn't specified, the label's desired height is computed from the size of the bitmap or text being displayed in it.

pad-x

specifies a non-negative value indicating how much extra space to request for the widget in the X-direction. When computing how large a window it needs, the widget will add this amount to the width it would normally need (as determined by the width of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space to the left and/or right of what it displays inside.

pad-y

specifies a non-negative value indicating how much extra space to request for the widget in the Y-direction. When computing how large a window it needs, the widget will add this amount to the height it would normally need (as determined by the height of the things displayed in the widget); if the geometry manager can satisfy this request, the widget will end up with extra internal space above and/or below what it displays inside.

relief

See section 1.2.2.

text

specifies a string to be displayed inside the widget. The way in which the string is displayed depends on the particular widget and may be determined by other pseudo-slots, such as *anchor* or *justify*.

text-variable

specifies the name of a variable. The value of the variable is a text string to be displayed inside the widget; if the variable value changes then the widget will automatically update itself to reflect the new value. The way in which the string is displayed in the widget depends on the particular widget and may be determined by other pseudo-slots, such as *anchor* or *justify*. See below how to use this mechanism.

width

Specifies a desired width for the label. If a bitmap is being displayed in the label then the value is in screen units; for text it is in characters. If this pseudo-slot isn't specified, the label's desired width is computed from the size of the bitmap or text being displayed in it.

2.3 The Message class

The *Message* class defines widgets that display a textual string. A *message*widget has three special features. First, it breaks up its string into lines in order to produce a given aspect ratio for the window. The line breaks are chosen at word boundaries wherever possible (if not even a single word would fit on a line, then the word will be split across lines). Newline characters in the string will force line breaks; they can be used, for example, to leave blank lines in the display.

The second feature of a message widget is justification. The text may be displayed left-justified, centered on a line-by-line basis, or right-justified.

The third feature of a message widget is that it handles control characters and non-printing characters specially. Tab characters are replaced with enough blank space to line up on the next 8-character boundary. Newlines cause line breaks. Other control characters (ASCII code less than hexadecimal code 20) and characters not defined in the font are displayed as a four-character sequence
x*hh* where *hh* is the two-digit hexadecimal number corresponding to the character.

A message object has several pseudo-slots which are listed here:

anchor

(see section 2.2.2)

aspect

specifies a non-negative integer value indicating desired aspect ratio for the text. The aspect ratio is specified as 100*width/height. 100 means the text should be as wide as it is tall, 200 means the text should be twice as wide as it is tall, 50 means the text should be twice as tall as it is wide, and so on. Used to choose line length for text if *width* pseudo-slot isn't specified.

background

(see section 1.2.2)

border-width

(see section 1.2.2)

cursor

(see section 1.2.2)

font

(see section 2.2.2)

justify

specifies how to justify lines of text. Must be one of *:left*, *:center*, or *:right*. Defaults to *left*. This pseudo-slot works together with the *anchor*, *aspect*, *padX*, *padY*, and *width* pseudo-slots to provide a variety of arrangements of the text within the window. The *aspect* and *width* pseudo-slots determine the amount of screen space needed to display the text. The *anchor*, *padX*, and *padY* pseudo-slots determine where this rectangular area is displayed within the widget's window, and the *justify* pseudo-slot determines how each line is displayed within that rectangular region. For example, suppose *anchor* is *:e* and *justify* is *:left*, and that the message window is much larger than needed for the text. The text will displayed so that the left edges of all the lines line up and the right edge of the longest line is *pad-x* from the right side of the window; the entire text block will be centered in the vertical span of the window.

pad-x

(see section 2.2.2)

pad-y

(see section 2.2.2)

relief

(see section 1.2.2)

text

(see section 2.2.2)

text-variable

(see section 2.2.2)

width

specifies the length of lines in the window. If this pseudo-slot has a value greater than zero then the *aspect* pseudo-slot is ignored and the *width* pseudo-slot determines the line length. If this pseudo-slot has a value less than or equal to zero, then the *aspect* pseudo-slot determines the line length.

2.4 The Button class

The *Button* class defines widgets that display a "reactive" textual string or bitmap. It can display itself in either of three different ways, according to the *state* pseudo-slot; it can be made to appear raised, sunken, or flat; and it can be made to flash. When a user invokes the button (defaultly, by pressing mouse button 1 with the cursor over the button), then the command specified in the *command* pseudo-slot is invoked. Several methods are also defined for buttons. They are presented at end of chapter.

A button object has several pseudo-slots which are listed here:

active-background

specifies background color to use when drawing active elements. An element (a widget or portion of a widget) is active if the mouse cursor is positioned over the element and pressing a mouse button will cause some action to occur.

active-foreground

specifies foreground color to use when drawing active elements. See above for definition of active elements.

anchor

(see section 2.2.2)

background

(see section 1.2.2)

bitmap

(see section 2.2.2)

border-width

(see section 1.2.2)

command

specifies a command to associate with the button. This command is typically invoked when mouse button 1 is released over the button window.

cursor

(see section 1.2.2)

disabled-foreground

specifies foreground color to use when drawing a disabled element. If the pseudo-slot is specified as an empty string (which is typically the case on monochrome displays), disabled elements are drawn with the normal foreground color but they are dimmed by drawing them with a stippled fill pattern.

font

(see section 2.2.2)

foreground

(see section 2.2.2)

height

specifies a desired height for the button. If a bitmap is being displayed in the button then the value is in screen units; for text it is in lines of text. If this pseudo-slot isn't specified, the button's desired height is computed from the size of the bitmap or text being displayed in it.

pad-x

(see section 2.2.2)

pad-y

(see section 2.2.2)

relief

(see section 1.2.2)

state

specifies one of three states for the button: *:normal*, *:active*, or *':disabled*. In normal state the button is displayed using the *foreground* and *background* pseudo-slots. The active state is typically used when the pointer is over the button. In active state the button is displayed using the *activeForeground* and *activeBackground* pseudo-slots. Disabled state means that the button is insensitive: it doesn't activate and doesn't respond to mouse button presses. In this state the *disabledForeground* and *background* pseudo-slots determine how the button is displayed.

text

(see section 2.2.2)

text-variable

(see section 2.2.2)

width

Specifies a desired width for the button. If a bitmap is being displayed in the button then the value is in screen units;for text it is in characters. If this option isn't specified, the button's desired width is computed from the size of the bitmap or text being displayed in it.

Example The following example show a simple use of buttons.

```
(setq b1 (make-instance 'Button :text "Yes"
                                 :command '(format t "You said Yes")))
(setq b2 (make-instance 'Button :text "No"
                                 :command '(format t "You said No")))
(event-loop)
```

2.5 Functions of this chapter

Chapter 3

Simple widget methods

3.1 pack

3.2 raise

3.3 lower

Chapter 4

The Canvas widget

4.1 The ¡Canvas¿ class

4.2 The ¡Canvas-item¿ class

4.2.1 ¡Rectangle¿ class

4.2.2 ¡Oval¿ class

......
......
......

4.3 Grouping canvas item objects

...

Chapter 5

The Text widget

5.1 The ¡Text¿ class

5.2 The ¡Text-tag¿ class

5.3 The ¡Text-mark¿

Chapter 6

Composite widgets

6.1 Standard composite widgets

6.1.1 Labeled Entry

6.1.2 Default button

6.1.3 Choice button

6.1.4 Paned window

6.1.5 Scrool Listbox

6.1.6 Scroll Text

6.1.7 Scroll Canvas

6.1.8 File box

6.2 Writing composite widgets

Index