STklos : A Scheme Object Oriented System Dealing with the Tk toolkit

Erick Gallesio
Université de Nice - Sophia-Antipolis
Laboratoire I3S - CNRS URA 1376 - ESSI.
Route des Colles
B.P. 145
06903 Sophia-Antipolis Cedex - FRANCE

email: eg@unice.fr

Abstract

STk is a graphical package which rely on the Tk toolkit and the Scheme programming language. Concretely, it can be seen as the Tk package where the Tcl language as been replaced by a Scheme interpreter. STklos is an object oriented extension of STk. Usage of this object extension facilitates code reuse and the definition of new widgets classes.

1 Introduction

Today available graphical toolkits for applicative languages are often unsatisfactory. Most of the time, they ask to the user to be a GUI expert who must cope with details such as server connections or queue events which are far lower than the concepts such languages vehicle.

Among all the graphical toolkits available in the X world, the Berkeley Tk package developed by John Ousterhout merits a great attention [Ousterhout 94]. This toolkit provides to the user high level widgets such as buttons, menus or text editors which permit to build complex GUIs with little effort. In particular, a little knowledge of X fundamentals is needed to build a complete running application with it. Tk package relies on a simple interpretative language named Tcl [Ousterhout 90]. This language is a string based language with a shell-like syntax. If the Tcl language is convenient for small scripts writing, its usage in bigger projects is not suitable, because it lacks important features such as data structures, types and objects. Furthermore, its imperative nature complicates the study of some interesting paradigms such as prototypes, actors or objects for GUI programming.

All these reasons have conducted to the definition of the STk graphical package, a package based on Tk where the Tcl language as been replaced by a Scheme [Clinger 91] interpreter. Usage of an applicative language will permit to simplify the implementation of new programming paradigms. For now, only the object paradigm has been implemented on the STk platform. This object oriented extension, which is called STklos, provides objects *à la* CLOS (Common Lisp Object System) [Steele 90]. More precisely, STklos is much closer from the objects one can find in Dylan [Apple 92], since this language is already a tentative to merge CLOS objects notions in a Scheme like language.

The STklos extension gives to the user a full object oriented system with meta-classes, multi-inheritance, generic functions and multi-methods. Furthermore, all the implementation rely on a true meta object protocol, in the spirit of the one defined for CLOS [Kickzales 91]. This model has been used to embody the predefined Tk widgets in a hierarchy of STklos classes. This set of classes permits to simplify the core Tk usage by providing homogeneous accesses to widget options and by hiding the Tk widget low level details, such as naming conventions. Furthermore, as expected, usage of objects facilitates code reuse and the definition of new widgets classes.

This paper is divided in three sections. Next section presents the STk package and its object system. Integration of the standard Tk widgets in STklos classes is described in the following section. Last section is devoted to the definition of composites widgets.

2 Presentation of STklos

Programming with STk can be done at two distinct levels. First level uses only the standard Scheme constructs and is classical. Second level gives access to the object oriented extensions of STk and is far more interesting. Of course, both levels can be used at the same time in a program if needed.

2.1 STk: the basic layer

Starting a session with the STk interpreter brings the user in the basic layer which gives him/her access to a complete Scheme interpreter extended to deal with the Tk toolkit. With a little set of rewriting rules from the original Tcl-Tk library, and the Tk manual pages close at hand, one can easily build a STk program using the Tk toolkit.

Creation of a new widget (button, label, canvas, ...) is done with special STk primitives procedures. For instance, creating a new button can be done with

```
(button '.b)
```

Note that the name of the widget must be *quoted* due to the Scheme evaluation mechanism. The call of a widget creation primitive, such as `button` above, defines a new Scheme object which is called a *Tk-command*. This object, which is considered as a new basic type by the STk interpreter, is automatically stored in a variable whose name is equal to the symbol passed to the creation function. So, the preceding button creation would define an object stored in the `.b` variable. This object is a special kind of function which is generally used, as in pure Tk, to customize its associated widget. For instance, the expression

```
(.b 'configure
    '-text "Hello, world"
    '-border 3)
```

permits to set the text and background options of the `.b` button. As we can see on this example, parameters must be well quoted in regard with the Scheme evaluation rules. Since this notation is barely crude, the Common Lisp keyword mechanism has been introduced in the Scheme interpreter [1]. Consequently, the preceding expression could have been written as

```
(button '.b
        :text "Hello, world"
        :border 3)
```

The Tk binding mechanism, which serves to create widget event handlers follow the same kind of rules. The body of a Tk handler must be written, of course, in Scheme. Following example shows such a script; here, the label `.lab` indicates how many times mouse button 3 has been depressed over it. Increment of button press counter is achieved with the simple script given in the `bind` call.

```
(define count 0)
(label ´.lab :textvariable 'count)
(bind .lab "<ButtonPress-3>"
      '(set! count (+ count 1)))
```

Programming with this kind of constructions is a little bit tedious and more complicated than coding with Tcl since we have to add parenthesis pairs and quote options values. Its only interest is to bring the power and flexibility of the Tk world to an applicative language.

2.2 STk: the object layer

STk provides an object extension, called STklos, which can be loaded dynamically. Using this object layer permits to gain the benefits inherent to object oriented programming. Furthermore, since STklos implementation rely on a meta-object protocol, it is easy to adapt it to particular needs as we will see in next section. Before that, we'll present here the main constructs available in this object layer independently of the Tk toolkit.

Definition of a new class is done with the `define-class` macro. For instance,

```
(define-class Point
  ((x :init-keyword :x
      :accessor     x-of)
   (y :init-keyword :y
      :accessor     y-of))
```

1. A keyword is a symbol beginning with a colon. It can been seen as a symbolic constant (i.e. its value is itself).

defines a point characteristics. Two slots are declared: x and y. A set of options can be expressed in a slot definition. Here, for instance, it is said that both slots can be initialized upon instance creation with the keywords :x and :y. Furthermore, it is asked to the system to generate an *accessor* function for each slot.

Creation of a new instance is done with the make constructor:

```
(define p (make Point :x 10 :y 20))
```

Evaluation of preceding form permits to build a new point and to initialize its slots x and y to 10 and 20.

Reading the value of a slot can be done with the function slot-ref. For instance,

```
(slot-ref p ´x)
```

permits to get the value of slot x in the p point. Setting a slot can be done by using the function slot-set!. For instance, setting the y slot of p can be written:

```
(slot-set! p ´y 0)
```

Since the accessor y-of has (automatically) been defined on this slot, its value can also be retrieved with the following expression:

```
(y-of p)
```

Slot setting can be done with the generalized set! defined in STklos:

```
(set! (y-of p) 1)
```

In STklos, execution of a method doesn't rely on the classical message sending mechanism as in numerous object languages but on *generic functions*. The mechanism implemented in STk is a subset of the CLOS generic functions. As in CLOS, a generic function can have several methods associated with it. These methods describe the generic function behavior according to the type of its parameters. For instance, the x-of accessor defined before is implemented via a generic function. It means that we can have several methods whose name is x-of. However, calling x-of with a parameter which is an instance of a Point will always call the accessor defined before.

3 Integration of Tk widgets

3.1 The class hierarchy

This section presents how the standard Tk widgets have been embodied in STklos classes. In this package, every type of graphical object defined by the Tk toolkit such as menu, label or button widgets is represented as a STklos class. All these classes defined for the Tk toolkit constitute a hierarchy which is briefly described here. Firstly, all the classes share a unique ancestor: the <Tk-object> class. This class defines a set of informations which are necessary to establish a communication between the Scheme and Tk worlds. In particular, three slots are defined in this class:

- The parent slot contains a reference to the object which (graphically) includes the current object.
- The Id slot contains, the low level *Tk-command*, generated by the system to implement the widget. The type of this *Tk-command* is different for each class. This slot is heavily used by the methods which implement the behavior of the object.
- The Eid slot contains the *Tk-command* which permit to manipulate the object from the outside (e.g. to map the object on the screen or to destroy it). For simple Tk widgets, Id and Eid always contain the same *Tk-command*.

Normally, end users will not have to use direct instances of the <Tk-object> class[2].

The next level in our class hierarchy defines 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* or *menus* since instances of the latter are objects contained in a canvas such as *lines* or *rectangles*. All those widgets are directly implemented as STklos objects in a one-to-one relationship. A partial view of the STklos hierarchy is shown in Figure 1.

Some points are important to note here:

- There is no difference between items of a can-

2. All classes whose name begins with the "Tk-" prefix are not intended for the final user.

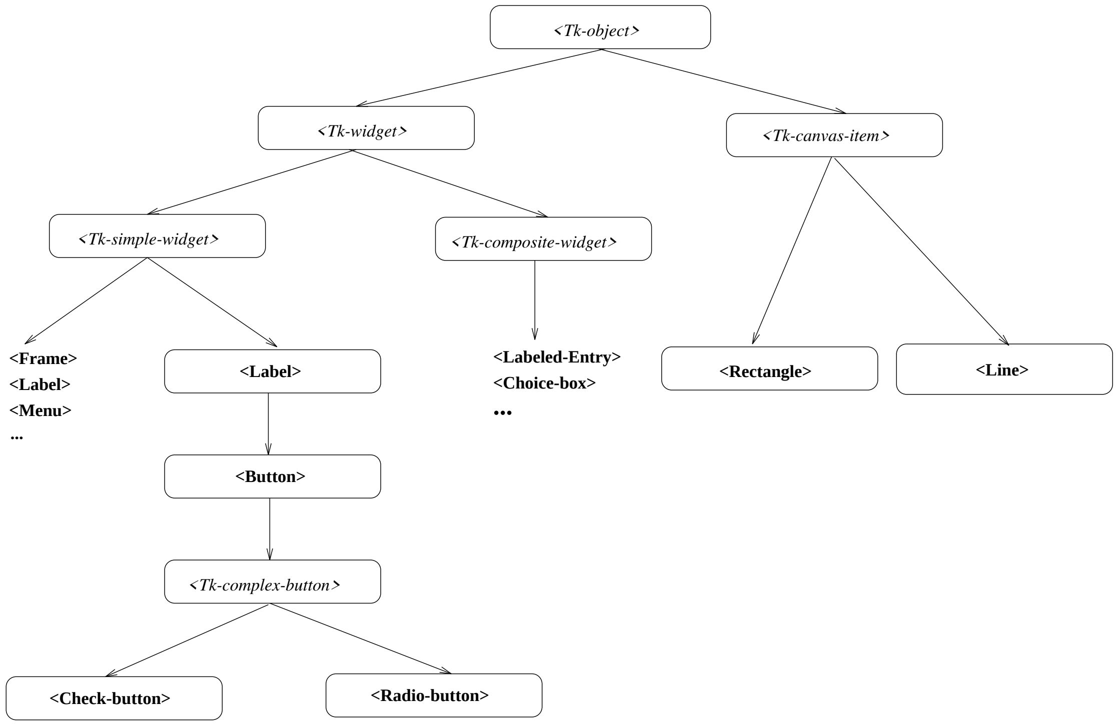

Figure 1: A partial vue of STk hierarchy

vas (e.g. a rectangle) and interface widgets (e.g. a label). This is an important difference with the Tk view of canvas items. In Tk, manipulation of a canvas item requires a reference to the canvas which contains it. In STklos the object itself knows the canvas to which it belongs and it can be forgotten when manipulated.

• Some simple widgets are already obtained by inheritance. For instance, a *button* can be seen as a reactive *label*. This permits to group all the methods to manage the appearance of the text of a label in the `<Label>` class. Thus, the `<Button>` class has only to manage the operations which are specific to reactive texts.

• Simple and composites widgets share a common ancestor (`<Tk-widget>`). This will permit us to define composite widgets which could be controlled exactly as the toolkit built-in widgets.

3.2 Accessing widgets options

In STklos, each option of a Tk widget is seen as an object slot. For instance, a simplified definition of a Tk button could be:

```
(define-class <Button>
             (<Label>)
   ((command :accessor     command
             :init-keyword :command
             :allocation   :pseudo))
   :metaclass <Tk>)
```

This declaration permits to define the class `<Button>`. This class inherits from `<Label>` and owns an extra slot called `command`. Allocation of this slot is said to be `:pseudo`. Pseudo-slots are special purpose slots: they can be used as normal slots but they are not allocated in the Scheme world (i.e. their value is stored in one of the structures manipulated by the Tk library instead of in a Scheme object). Consequently, reading or writing this slot will be done in a particular way[3]. Pseudo slots are implemented by the meta-class `<Tk>`. Defining a class using this meta-class permits to modify the protocol to access a slot at its lowest level. Consequently, it is impossible to have a slot value which does not reflect the actual value of the Tk option (remember that no space is reserved to save the value of this slot in the

3. Access to a pseudo slot will be done using the *configure* sub-command which is available for each Tk widget.

Scheme space). Saying that the meta-class of the `<Button>` class is `<Tk>` (the one which knows what to do with pseudo slots) is done with the `:metaclass` option. With this definition, the system is able to build slot readers and writers which take into account pseudo slots. It is important to note that the accessors construction is done at class creation. Consequently, no test is done when accessing a slot to know what kind of allocation it uses.

Embodying Tk widgets in STklos objects permits to hide some Tk idiosyncrasies which, in turn, improves greatly the level of programming when building an interface. In particular, it avoids the knowledge of pure Tk widgets naming convention which is a pain when developing large applications. The only thing the user has to know when creating a new object is it's parent. An example of widgets creation is shown below:

```
(define f (make <Frame>))

(define b1 (make <Button>
                 :text "B1"
                 :parent f))

(define b2 (make <Button>
                 :text "B2"
                 :parent f)
```

Buttons `b1` and `b2` which are created here specify that their parent is the frame `f`. Since this frame does not specify a particular parent, it is supposed to be a direct descendant of the root window `*root*`. This parent's notion is also used for canvas items: a canvas item is considered as a descendant of the canvas which contains it. For instance,

```
(define c (make <Canvas>))

(define r (make <Rectangle>
                :parent c
                :coords '(0 0 50 50)))
```

defines a rectangle called `r` in the `c` canvas. User can now forget that `r` is included in `c` since this information is embedded in the Scheme object which represent it. For instance, the following expression permits to move the `r` rectangle without having to cite the `c` canvas.

```
(move r 10 10)
```

Similarly, the expression

```
(bind r "<Enter>" '(display "Hello\n"))
```

permits to display a message each time the mouse enters in the `r` rectangle. It is important to note here that we would use *exactly* the same expression to associate such a binding to a simple widget such as a button or a label, whereas it takes two different syntactic forms using standard Tk.

Usage of generic functions is also a significant improvement over the basic level since it allows an homogeneous access to the Tk commands. Suppose for instance that we want giving access to the value of a `scale` or an `entry` widget with the generic function `value-of`. This can easily done by the following methods in STklos:

```
(define-method value-of (obj)
   ((slot-ref obj 'Id) 'get))
```

In this case, one method is sufficient to implement our function since the sub-options for reading a scale or entry value is the same in Tk. Writing this value is a little bit more complicated and is given below:

```
(define-method
     (setter value-of)((obj <Scale>) v)
   ((slot-ref obj 'Id) 'set v))

(define-method
     (setter value-of)((obj <Entry>) v)
  ((slot-ref obj 'Id) 'delete 0 'end)
  ((slot-ref obj 'Id) 'insert 0 v))
```

Using the same generic function (with two different methods) permit to hide these low level details. In the call,

```
(set! (value-of x) 100)
```

the system will choose the method to call depending of the actual type of the variable `x`. Furthermore, an error[4] will be signalled if `x` is not an *entry* or a *scale*.

STklos also proposes the notion of *virtual slot*. Virtual slots requires no storage as the pseudo slots seen before. It is to the programmer to define methods to retrieve and store the value of such a slot. Virtual slots permit to easily implement the previous `val-`

4. more exactly, the system calls the `no-applicable-method` generic function which, by default, signals an error. User can specialize this function to provide another handler if needed

Figure 2: A labeled entry

ue-of notion directly as a slot, rather than a generic function. Virtual slots will be not described here.

As we have seen, programming with STklos brings the power of a full object language to the process of graphical user interface building. Next section will show that it also permits to easily implement composite widgets.

4 Defining composite widgets

Today, object oriented programming languages have proven their usefulness in program construction. In particular, it is a cliché to say that they facilitate program maintainability and code reuse. However, there is one point where object oriented paradigm is not so well suited: creation of new classes of objects which are the composition of simpler ones.

4.1 The problem

To illustrate our purpose, let us have a look to a *labeled entry*. A labeled entry is a small line editor with a label on its left informing the user about the value he/she is supposed to provide (see Figure 2).

Since this kind of graphical object does not exist per se in pure Tk, we could want to create a new class for labeled entries. In Tk, this kind of object can be implemented by composing three basic objects: a *label*, an *entry* and a *frame* which will group them. Once this composition is done, the frame will serve to manipulate the labeled entry from the outside (to place it on the screen for instance). Of course, accessing to the text entered by the user or to the label of this object will necessitate to "open" the frame.

4.2 Classical solutions

Let us look how we could implement a class <Labeled-entry> for the kind of labeled entries specified above. The first approach consists to uses the multiple-inheritance mechanism to implement this new class. In this case, we can inherit from the predefined classes <Frame>, <Label> and <Entry>. Unfortunately, this will not work since inheritance will share all the common slots. It means, for instance, that the slot parent which contains the parent widget of a graphical object, and which is present in all the components of this new object, will not be duplicated. Having a sole exemplary of this slot is a problem since those components don't share the same parent (label and entry parent is the frame itself). It could be argued that a different slot inheritance scheme should resolve the problem. For instance, it would be possible to duplicate common slots instead of sharing them, as it is possible in Eiffel for instance [Meyer ??]. This solution would probably simplify our problem. However, the methods already written for a component of our new kind of entry will be in front of three slots with the same name. In this case, we can decide whose slot is the more appropriate but it would be less clear if two components were of the same type.

Since multiple inheritance is helpless here, we can try to use single inheritance to resolve this problem. In this case, we have to choose the more adequate class from which inherit, and other component will be stored in new defined slots. Inheriting from the <Entry> class is clearly more accurate since the behavior expected from a labeled entry is close from a single entry behavior. In some other situation this choice could be less obvious. Here, getting the content of the entry part can be done by calling the generic function value-of described in previous section. With the proposed inheritance scheme, a call to value-of whose argument is a <Labeled-entry> will call the method defined for single entries (i.e. which are instances of the <Entry> class). Provided that the slot Id of the labeled entry contains the *Tk-command* used to implement the entry, this call will yield the correct value.

However, keeping in a slot the entry identification does not permit to manipulate the labeled entry as an autonomous entity[5]. This problem can be easily solved: it suffices to introduce a slot in the root class for global widget manipulations. This slot, called Eid, will always contain a reference to the most "external" graphical object of our widget. This slot is set to the value contained in the Id slot for simple widget as said before; it is set generally to the

5. For instance, this is the *frame* identification which is needed to destroy all the components of a labeled entry rather than the entry one.

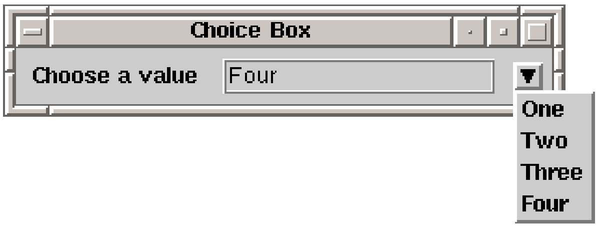

Figure 3: A Choice Box

frame's `Id` which embodies the components of a composite widget.

This implementation of the labeled entry is not yet completely satisfactory. With the previous inheritance scheme, the entry sub-widget plays a major role in the labeled entry. In some occasions, however, we could want that the accesses to a particular slot will be "redirected" to another sub-widget. For instance, we could want that readings and writings to the slot `relief` of a labeled entry access in fact to the relief of the frame rather than the entry relief. Eventually, we could also want that a slot modification will be propagated to several sub-widgets of the labeled entry. For instance, it would be suitable to propagate the modification of the `background` slot of a labeled entry not only to the entry but also to the label and the frame. What is wanted here is close to the delegation mechanism[Libermann 86]. As we will see, the solution provided in STklos will permit to choose to which sub-widget(s) a slot access must be redirected.

4.3 The <Tk-composite> class

STklos provides a special meta-class, named `<Tk-composite>` for handling the creation of composites widgets. The main job of this meta-class consists to manage a special kind of slots whose access can be redirected to slots of other objects. Using this meta-class, a simple implementation of the labeled entry discussed before could be written as:

```
(define-class <Labeled-entry>
        (<Tk-composite-widget> <Entry>
  ((entry         :accessor      entry-of)
   (label         :accessor      label-of)
   (background    :accessor      background
                 :init-keyword :background
                 :allocation   :special
                 :propagate    (frame entry
                               label))
   (value         :accessor      value-of
                 :init-keyword :value
                 :allocation    :special
                 :propagate     (entry)))
:metaclass <Tk-composite>)
```

As we can see, a label entry inherits from the `<Entry>` class. This class defines all the slots (i.e. Tk options) available for an entry. It is important to note here that even if the slots for such a class are numerous, only three of them are effectively allocated in a standard Tk widget (namely `parent`, `Id` and `Eid`). All the other slots are pseudo-slots which are allocated elsewhere[6].

Previous class definition augments the `<Entry>` class with three new slots called `frame`, `entry` and `label`. These slots will contain the sub-widgets composing the labeled entry. Next, two slots are declared with a *special* allocation protocol (signaled with the `:special` keyword). Special slots are slots for which reading and writing are redirected to other sub-widgets. Here again, such slots are not directly implemented in an instance; they don't make the instance size growing. The `background` slot of this class definition, for instance, states that setting its value must be propagated to the entry, label and frame slots (reading of this slot will find its value in the first element of the `:propagate` list: frame). Note that `background` is already present in the `<Entry>` as a pseudo-slot; current definition will overload the inherited one. Concerning the `value` slot, it is said that it is redirected only to the slot `value` of the `entry` component.

We can now define the `initialize-composite-widget` method which will be called by the system just after a composite widget allocation (complete code is given in annex). The only thing we have to do in this method consists to initialize the two true slots defined directly in the `<Labeled-entry>` class (`entry` and `label`).

Previous definitions permits to hide to the user the fact that a labeled entry is a composite widget by offering the same kind of access. For example, suppose now that `le` denotes an instance of the class `<Labeled-entry>`; setting all its components to `¨grey¨` can be simply done by

```
(set! (slot-value le 'background)
      ¨grey¨)
```

or

```
(set! (background le) ¨grey¨)
```

6. `<Tk-composite-widget>` defines also a slot named `frame` which contains the external frame of the composite widget.

This call will activate the setter function of the `background` slot. This setter function will do the three slot assignments needed to change totally the background. Note that this setter function is created only once in the program (i.e. at the `<Labeled-entry>` creation time). Consequently, the only overhead we have to access such a special slot (*vs.* a true-widget pseudo slot) is the call to the initial `slot-value`.

In the object oriented programming spirit, our composition mechanism must be applicable more than once. It means that we must be able to build new composite widgets which are built upon previously created composite widgets. Consider for instance a *choice box* widget. This new kind of widget could be implemented with a labeled entry, as the one we have used until now, to which is associated a menu button giving a list of possible choices (see Figure 3). Here, the `<Choice-box>` inherits of the previous `<Labeled-Entry>` class and a possible definition for this class could be:

```
(define-class <Choice-Entry>
             (<Labeled-Entry>)
  ((frame      :accessor frame-of)
   (lab-entry  :accessor lentry-of)
   (menu       :accessor menu-of)
   (menubutton :accessor menubutton-of)
   (value      :accessor      value
               :init-keyword :value
               :allocation   :special
               :propagate    (lentry))

  :metaclass <Tk-composite>)
```

This simple class definition and its associated `initialize` method suffice to define a choice box. We can see that access to the `value` special slot will still give a correct value. For instance, getting this slot will redirect the reading to the `lab-entry` slot which in turn will redirect it to the reading of the slot `value` of the entry, as seen before.

5 Conclusion

In this paper we have presented the STk interpreter and its object oriented extension. Both packages are well integrated with the Tk graphical toolkit. Even if using an applicative or object oriented language for GUI programming is not a new idea ([Calder 87], or [Garnet] for instance), rare are those which use extensively a meta object protocol. Investigating in this direction seems interesting and the results already obtained are promising. In particular, the original mechanism of composition shown in this paper is a good illustration of adapting a meta protocol to a particular need. It permits to build and test new widgets without having to recompile the toolkit. Furthermore, daily experience of STk and STklos show that an applicative object oriented languages can be comparable, in terms of performances, to a more classical language.

Availability

STk and its object system STklos are distributed free of charge by anonymous ftp at `kaolin.unice.fr`. Current version runs on

- Sun Solaris (1 & 2),
- Ultrix (4.2),
- Dec OSF1,
- SGI (Irix4.05 & 5.1.1)
- Linux(0.99).

References

[Apple 92] Apple, *Dylan: an object oriented dynamic language*, Apple Computer, 1992.

[Calder 87] Mark A. Linton, Paul R. Calder and John Vlissides, *The Design and Implementation of InterViews,* Proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico,november 1987.

[Clinger 91] W. Clinger and J. Rees (editors), *Revised^4 Report on the Algorithmic Language Scheme*, ACM Lisp Pointers, 4 (3), 1991.

[Kickzales 91] Gregor Kickzales, Jim de Rivières, Daniel G. Bobrow, *The Art of Meta Object Protoco*l, MIT Press, 1991.

[Lieberman 86] Henry Lieberman, *Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object Oriented Systems*, Actes des 3^e JLOO, Bigre+Globule, 48, pages 79-89, Paris, 1986.

[Meyer 89] Bertrand Meyer, *Object Oriented Software Construction*, Prentice Hall International (UK), Ltd., Hemel Hemstead, 1989.

[Myers 89] Brad Myers, *The Garnet Toolkit Reference Ma-*

nuals: Support for Highly Interactive, Graphical User Interfaces in Lisp, Technical Report, CMU-CS-89-196, Pittsburg, PA, 1989.

[Ousterout 90] John K. Ousterhout, *Tcl: an embeddable command language*, USENIX Winter Conference, January 1990, pages 183-192.

[Ousterout 94] John K. Ousterhout, *Tcl and the Tk toolkit*, Addison-Wesley, 1994, ISBN: 0-201-63337-X.

[Steele 90] Guy. L. Steele Jr., *Common Lisp: the Language*, 2nd Edition, Digital Press (Bedford, MA), 1990.

Annex

Complete code of the **<Labeled-entry>** class.

```
;;;; Define class "<Labeled-entry>"

(define-class <Labeled-entry> (<Tk-composite-widget> <Entry>)
  ((entry  :accessor   entry-of)
   (label :accessor   label-of)
   ;; Special slot
   (text       :accessor     text
               :init-keyword :text
               :allocation   :special
               :propagate    (label))
   (value      :accessor     value
               :init-keyword :value
               :allocation   :special
               :propagate    (entry))
   (background  :accessor     background
               :init-keyword :background
               :allocation   :special
               :propagate    (frame entry label)))
  :metaclass <Tk-composite>)

;;;; Define method "initialize-composite-widget". This method will be called when
;;;; a new labeled entry will be created.

(define-method initialize-composite-widget
                    ((self <Labeled-entry>) initargs frame)

  (let* ((e (make <Entry> :parent frame :relief "ridge"))
         (l (make <Label> :parent frame))

    ;; pack sub widgets
    (pack l :side "left"  :padx 5 :pady 5)
    (pack e :side "right" :padx 5 :pady 5 :expand #t :fill "x")

    ;; Set proper slots
    (slot-set! self 'Id     (Id e))
    (slot-set! self 'entry  e)
    (slot-set! self 'label  l)))

;;;; To create the labeled entry of Figure 2:
(define le (make <Labeled-entry> :text "Enter a value" :value 100))

;;;; To map it on the screen:
(pack le)
```