A Framework for Visualizing Object-Oriented Systems

Volker Haarslev*
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304, USA
haarslev@parc.xerox.com

Ralf Möller
University of Hamburg, AI Laboratory
Bodenstedtstr. 16, D-2000 Hamburg 50, FRG
moeller@rz.informatik.uni-hamburg.dbp.de

Abstract

This paper describes a new approach to visualizing program systems within the object-oriented paradigm. This approach is based on a TEX-like notation which has been extended and generalized for specifying graphical layout of arbitrary objects. The CLOS meta-level architecture is used to associate visualization and application objects. We propose several useful techniques such as indirect values, slot and method demons, and instance-specific meta-objects. Our techniques require no modifications to the systems which are selected for visualization. We demonstrate the feasibility of our approach using application domains such as CLOS debugging and constraint systems.

1 Introduction

Although programming has mostly been done in textual terms users have always had a notion of visualizing their programs. Programs have been entered as lines of text but soon users started to indent their programs and also used comments for separating or emphasizing particular program parts. Tools were developed which pretty-print or format source code. Modern programming environments offer debugging tools such as browsers and inspectors providing users with views of program structure and execution states. But these views display their information more textually than visually (pictorially). A further disadvantage of these environments is their lack of offering program designers adequate tools for visualizing and animating programs, which support both structural and conceptual visualization.

This paper discusses within the paradigm of object-oriented programming the use of structural and conceptual visualization techniques. We describe a new approach combining both techniques, which is based upon TEX-like layout specifications. Furthermore, we discuss the usefulness of meta-level architectures for implementing visualization techniques. We implemented a prototyping environment consisting of a set of extensible fundamental components which offer varying degrees of support. It is implemented in Macintosh Allegro Common Lisp and based upon the PCL implementation of the Common Lisp Object System [2, 17] (CLOS). We applied our visualization techniques to several application domains such as CLOS debugging, graphical editing, constraint systems, line routing, and concurrent logic programming (see also [24, 14]).

Conceptual visualizations of programs are mostly created by hand. This hand-design is basically caused by the fundamental problem that geometrical and graphical information necessary to create suitable visualizations cannot automatically be derived from corresponding data. The major problem is to define interesting events which should be visualized. But very often interesting events are only indirectly reflected by algorithms. We refer to [5] for a detailed discussion of these problems.

Structural visualization uses program and data structures to generate relevant geometrical information. An important problem related to structural interpretation is that conceptual information about data can only indirectly be derived (e.g. from naming of identifiers). A very common approach to structural visualization is to guide the visualization process by underlying programming styles or computational models. Many approaches to visualizing imperative systems use flow charts or diagrams. The Transparent Prolog Machine [9] is an example for relational or logic systems. A more radical approach is presented by Pictorial Janus [16]. It defines complete visualizations of concurrent logic programs and captures static as well as dynamic information about these programs.

There exist many approaches to visualizing data flow of functional systems, e.g. VIPEX [13], Pluribus [30], and Prograph [23, 6]. A diagramming approach to tracing object-oriented systems as an extension to a Smalltalk-80 debugger is described in [8]. GraphTrace [19] is also intended for understanding behavior of objects. It provides graphical traces of program executions. Both approaches are primarily focused on structural visualizations. In contrast to our approach they offer no support for conceptual visualizations.

Besides structural and conceptual visualization techniques it is also important to support flexible schemes for aesthetically laying out combinations of units. With respect to forms-oriented user interfaces allocation of space and position is mostly constrained by the space globally available. Graphical interfaces usually also add local constraints. A typical application is a browser generating net-like representations of rule sets, classes, or objects. The spatial allocation of nodes may depend on adjoining nodes or the topology of edges (e.g. in order to avoid line crossing or long winding paths). This problem is addressed by many constraint-oriented systems. ThingLab I [4] and II [22] are examples for describing layout of graphical objects with constraints. [27] also presented a toolkit using constraints

* New address: University of Hamburg, Computer Science Department, Bodenstedtstr. 16, D-2000 Hamburg 50, FRG, haarslev@rz.informatik.uni-hamburg.dbp.de

```
(let ((left-table (make-dialog-item ...))
      (right-table (make-dialog-item ...))
      (graph-view (make-layout-view ...)))
 (make-layout-dialog :layout
  (:vbox (:width :filler :height :filler)
   (:hbox (:height 1/4 :width :filler)
    (:fbox () left-table) (:fbox () right-table))
   (:fbox () graph-view)))
 (setf (layout graph-view) ...))
```

Figure 1: Layout specification of Figure 2 (schematically).

and active values. In contrast to constraint-oriented approaches we decided to provide a simpler but more compact and predictable notation for specifying layout. Furthermore, our approach has the advantage that it requires only $2n + \log(n)$ steps, be n the number of boxes. Thus, our algorithm has a computational complexity of $O(n)$ (see [15, 14] for details).

The remainder of this paper is structured as follows. The next two sections introduce our TeX-like specifications. Section 4 introduces box items which are suited for more general applications. The next section demonstrates a straightforward extension of a CLOS class browser which serves as an example for the flexibility of our approach. Afterwards we discuss the use of the CLOS meta-object protocol for program visualization and demonstrate some of these considerations using a simple constraint system as example. Section 7 compares our approach with related work. This paper concludes with a summary and a discussion of future work.

2 Layout Specifications

We adopted the "box-and-glue" metaphor of TeX [20] for specifying layout of objects. Layouts are composed of a set of rectangular regions, so-called *boxes*. Laying out boxes and positioning objects are associated with corresponding box types. Our system offers a set of predefined layout algorithms and box types. More general box types are discussed in Section 3.

2.1 Vertical and Horizontal Boxes

The fundamental scheme aligns boxes as a list of *horizontal* and *vertical* boxes. This layout technique has been found very useful for standard (forms-oriented) dialog windows (see Section 7 for a discussion of related work). A layout of a dialog window is specified as a combination of boxes with optional size specifications. Boxes may be arbitrarily nested. The size of boxes and the spatial relationship between them is expressed by an amount of *glue* or *filler* describing either a horizontal or vertical distance. Fillers can be specified as *fixed* (e.g. in pixel) or *variable*. Variable fillers depend on the space available to their enclosing box. We distinguish *relative* and *constrained* fillers. A relative filler is expressed as a fixed ratio to the size of its superior box. Constrained fillers can shrink (stretch) to a given lower (upper) limit. Default constraints are zero as lower limit and box size as upper limit. Several fillers as elements of the same box work together like springs. They share the available space and in general every filler claims

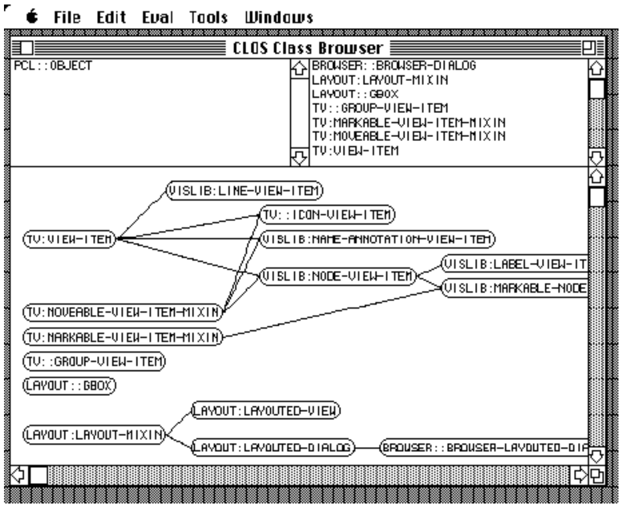

Figure 2: A DAG graph of a standard class hierarchy.

the same amount of space which is only constrained by its lower and upper limit.

A vertical or horizontal box (`<box-type>` either `:vbox` or `:hbox`) is specified by the lisp form `(<box-type> (:width h :height v) box-item-1 ...)`. Its box items are laid out vertically resp. horizontally. If the size specification is omitted the width and height of a vertical or horizontal box are set to a filler with default constraints. In general this layout algorithm keeps the size of box elements unchanged. If elements require more space than available to their surrounding box they are allowed to extend beyond their box's boundaries. Boxes are also allowed to overlap one another. But this behavior is not always desired. Therefore, we introduced a *frame box* (`:fbox`) which constrains the size of its element in order to match exactly the frame box size. A frame box contains only one box item: `(:fbox (:width h :height v) box-item)`.

2.2 Filler Specification

The complete form specifying a filler is `(:filler :min m :max n)`, `:min` and `:max` are optional. We defined `:filler` as short-form of `(:filler :min 0 :max box-size)`. It is also possible to define the (minimal/maximal) size of a box with respect to its elements. Then, the size of this box is set to the result achieved by laying out its elements and shrinking fillers to their lower limit (see [14] for more details). Thus, the box has a minimal size satisfying all lower bound constraints.

The Figures 1 and 2 show a layout specification and the resulting dialog window for a simple CLOS browser which displays a class hierarchy. The right table contains all direct subclasses of the class listed in the left table. The tables can be replaced by direct super resp. subclasses, scrolled, and shifted to focus on "interesting" classes. The lower part of the window displays a graph of the selected class hierarchy.

The browser dialog is specified as a vertical box with `:filler` as width and height. Its first item is a horizontal

```
(let ((upper-offset 10)
      (left-offset 10))
 (:vbox () upper-offset
  (:hbox () left-offset
   (:gbox
    (:dag
     *roots*
     #'successors
     *max-depth*
     #'appearance
     ...)))))
```

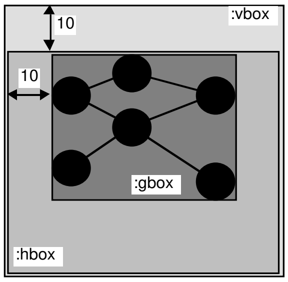

Figure 3: A general layout specification in combination with a box-style layout.[2]

box whose height is set to 1/4 of that of the vertical box. The horizontal box contains two scrollable tables which are enclosed by frame boxes. The height of these frame boxes is constrained by their surrounding horizontal box. Their width is not explicitly specified, therefore the default value `:filler` is chosen and half of the width of the horizontal box is assigned to each frame box (and its inferior table). The second item of the vertical box is a frame box surrounding the box element `graph-view` which generates the class graph. A layout form which is similar to that defining `graph-view` is shown in Figure 3.

3 Layout Protocols

Our basic layout algorithms are based on an abstract protocol for manipulating boxes and box items. Therefore, every object conforming to this protocol can be laid out and every (rectangular) region can be interpreted as a box. The protocol is implemented as a set of generic functions. Multi-object methods can be supplied for different kinds of boxes and items, which may represent position and size in different ways. The use of multi-object methods also has the advantage that layout of objects may depend on their context. Our layout protocol also allows to specify whether the layout algorithm has to be reapplied if global constraints (e.g. by resizing the window) have been changed.

Apparently it is not reasonable to describe every layout with the box-and-glue metaphor. An obvious example is a set of nodes arranged as a graph. Therefore, our layout language has the notion of a *general box*: `(:gbox (<layout-name> <arg-1>...<arg-n>))`. For instance, this box is used to specify the layout of directed acyclic graphs (DAGs) (see Figure 3).

In this example the items to be arranged are defined inductively by a set of roots, a successor function and a maximal depth. The appearance function is used to compute the graphical representation of nodes (e.g. class objects for an inheritance graph). Position and size of the general box (`:gbox`) are defined implicitly by a closure rectangle around all graph items (see Figure 3). This rectangle defines a box which may be arranged using the box layout specifications already known.

[2] The indentation of the boxes is used for demonstration purposes only.

One may also think of other arrangements of box items in a general box. We use the DAG example mentioned above in order to explain our protocol for supplying a new layout specification interpreter.[3]

```
(defmethod layout-spec-p-using-key
 ((key (eql ':dag))) t)
(defmethod parse-layout-spec-using-key
 ((key (eql ':dag)) layout-specs)
 "Returns (generated and) laid out :gbox items."
 (interpret-dag-layout layout-specs))
```

The layout name (e.g. `:dag`) of a general box form is used as a key to discriminate the corresponding layout interpreter method, the rest of the form is bound to the parameter `layout-specs`.

This extension scheme exploits that CLOS methods are not only attached to objects (or their classes) but can be also discriminated on every Lisp object. Layout forms are represented as lists, i.e. layout descriptions can easily be manipulated (e.g. by pattern matching algorithms).

4 Interaction Objects and Views

Interaction objects represent box items of layout specifications. Interaction objects (e.g. all standard elements of the Macintosh Toolbox, graph nodes, graph edges) are instances of CLOS classes. We defined an additional interaction object, a so-called *view*. Views provide a framework for handling non-standard interaction components. These components are called *view items*. View items can be freely added to and removed from views. The interactive behavior of view items can easily be modified by adding certain predefined superclasses (mixins) to their class definitions. Typical desired behaviors are to move, select, or mark items. The algorithms ensuring a consistent image on the screen are provided by views. Views can also be declared as scrollable.

The system evaluates generic functions to draw and delete visible items if required. Edges of graphs are also represented as view items. Edges use another feature of view items which is not subject to this paper: size and position of particular view items can be defined by referencing other view items. *References* are also specified using our box approach. For instance, the shape and position of each edge in the DAG view (Figure 2) is defined by referencing the nodes which are connected. We refer to [24, 15, 14] for more details.

5 Extended Class Browser

This section shows a slightly modified class browser. This version additionally displays direct slots of classes (see Figure 4). These extensions were easily achieved by defining appropriate methods for the generic functions `successors` and `appearance` (see layout form in Figure 3). These modifications serve as an example for the flexibility of our approach. A discussion of more sophisticated user interfaces is presented in [24].

[3] Layout forms are usually defined as macros evaluating the right expressions (in the right scope).

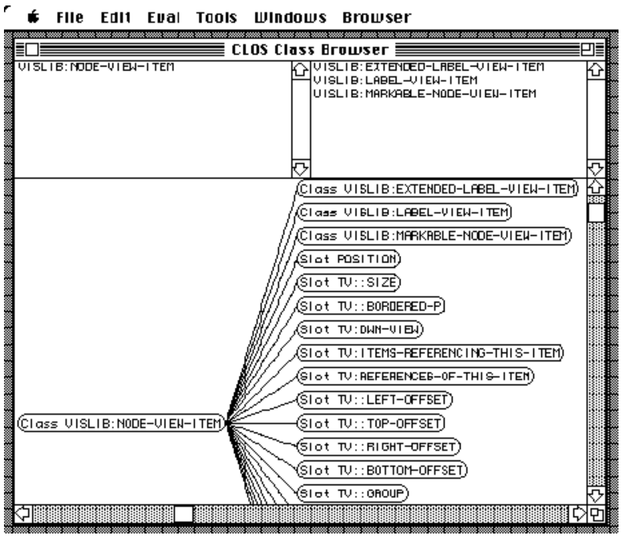

Figure 4: A DAG graph of a class hierarchy with local class slots.

```
(defmethod successors ((any t))
 "Do not show arbitrary objects"
 nil)
(defmethod successors ((class standard-class))
 "Show direct subclasses and slots"
 (append (class-direct-subclasses class)
         (class-direct-slots class)))
(defmethod appearance ((class standard-class))
 "Create a graphical class representation"
 (make-label class ...))
(defmethod appearance
 ((slot standard-slot-description))
 "Create a graphical slot representation"
 (make-label slot ...))
```

6 Meta-Level Techniques for Separating Application and Visualization

Application and visualization objects have to be separated. In the following we discuss how to use the meta-object protocol of CLOS for separating application and visualization layers. We explain these considerations by using an animated visualization of a simple constraint system as second example.

Our approach associates visualization objects with given application objects without requiring any modifications to the application. We support multiple views as well as controllers for manipulating the application's data structures. Several other mechanisms have been developed (Model-View-Controller-Scheme [11], CLUE [18], Presentation-Types [26]). In this section we also discuss how basic features of these systems can be realized using our approach.

As example application we chose a simple constraint net. There is no need to present the application code since everything can be found in detail in [29]. Our visualization was generated without any modifications to the application code. The application provides a simple model of a

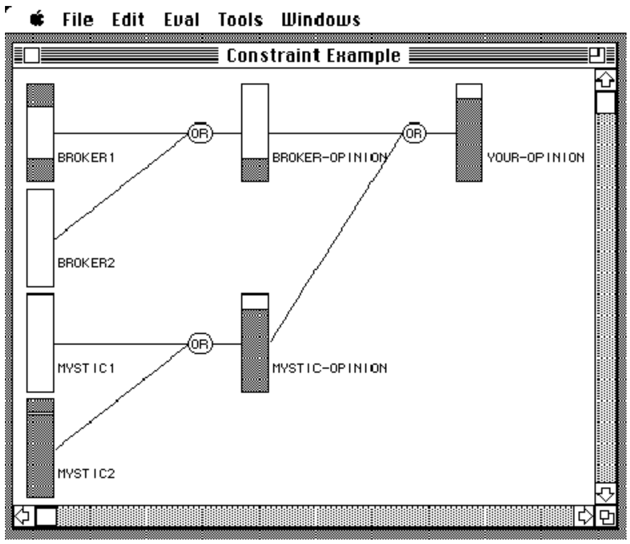

Figure 5: The upper and lower bounds are indicated by shaded rectangles [29]. The gauges show the estimation interval from 0 (bottom) to 1 (top).

stock exchange scenario. When are some stocks to split? The participants have uncertain knowledge and are influenced by one another. A constraint net models these influences by propagating certainty estimation intervals between 0 and 1. This interval of a 'broker' might be visualized by a gauge as found in [29]. The implementation distinguishes assertion objects (brokers, mystics, virtual intermediates, etc.) and constraint objects (or, and). Figure 5 shows an overview of an example configuration with gauges for assertions and simple nodes for constraints.

The visualization in Figure 5 can be described with the following layout descriptions.

```
(defun stock-exchange-connections-visualization
 (participants)
 "Opens a window and shows the connections of
  the given participants in a scrollable view."
 (let ((stock-exchange-view
       (make-layout-view
        :scroll-bars ':both :bordered-p nil
        :auto-scrolling t)))
  (make-stock-exchange-dialog
   (:fbox () stock-exchange-view))
  (setf
   (layout stock-exchange-view)
   (:vbox () 10
    (:hbox () 10
     (:gbox
      (:dag participants
            #'stock-exchange-wizard
            *max-connection-depth*
            #'application-visualization-coupler
            ...)))))))
```

The whole dialog consists of a view (laid out with an `:fbox`). The graph is defined by the set of participants and the successor function `stock-exchange-wizard`.

The function `application-visualization-coupler` defines a mapping from application objects to visualization objects. Both functions are generic, i.e. different mappings may be specified for different classes of application objects.

```
(defmethod stock-exchange-wizard
 ((participant assertion))
 "Wizard's inf. about conn. of assertion objects."
 (assertion-constraints participant))
(defmethod stock-exchange-wizard
 ((participant constraint))
 "Wizard's inf. about conn. of constraint objects."
 (list (constraint-output participant)))
```

6.1 Indirect Values for Visualization Objects

Visualization objects have to refer to objects of the application side. There should exist a "dynamic" binding which could be easily maintained provided that classes of visualization objects offer support for some kind of active values [3]. We present a simplified CLOS metaclass supporting non-nested active values which we call *indirect values*. Indirect values are defined by the form `` #`(object reader writer) `` where writer is optional. The following method sketches an implementation using a new metaclass and a corresponding meta-level method for the generic slot accessor function `slot-value-using-class`. Writing to slots with indirect values can be implemented analogously.

```
(defclass indirect-slots-class (standard-class) ())
(defmethod check-super-metaclass-compatibility
 ((x indirect-slots-class) (y standard-class))
 t) ; We do not care about that in this paper.4
(defmethod slot-value-using-class
 ((class indirect-slots-class) object slot-name)
 (let ((direct-slot-value (call-next-method)))
  (if (indirectp direct-slot-value)
   (funcall (indirect-reader direct-slot-value)
            (indirect-object direct-slot-value))
   direct-slot-value)))
```

Visualization objects have `indirect-slots-class` as metaclass. Using indirect slot values every slot access is delegated to the corresponding application object if required. Using the meta-object protocol it would be easy to determine all indirect objects or that indirect object referred to by a specific slot. The gauges for the stock exchange example use this metaclass to refer to the exchange participants. But what about the other direction: the gauges have to be "informed" if the participants' estimations of stock splits change.

6.2 Slot Demons for Application Objects

An assertion object has one slot for the lower bound and one for the upper bound estimation. The corresponding visualization objects have to be informed when either of these slot values change. The most obvious way to achieve this is to define the assertion class with a metaclass that allows *demon functions* to be attached to slots. The "real" value of a slot is a structure that provides a value facet and an `if-modified` facet [25]. All slot demon functions are evaluated when the slot value changes. The implementation of slot demons is similar to the one of indirect slot values. We introduce a metaclass `demon-slots-class` and define modified versions of `slot-value-using-class` and `(setf slot-value-using-class)` which access the value facet. The latter one evaluates the demons in the `if-modified` facet. Slot demons should be made removable.

Thus, the function `application-visualization-coupler` mentioned above can be defined as follows.

```
(defmethod application-visualization-coupler
 ((participant assertion))
 (let
  ((assertion-gauge
    (make-two-level-gauge   ; indirect values
     #`(participant assertion-lower-bound)
     #`(participant assertion-upper-bound))))
  (add-slot-if-modified-demon
   participant  ; object
   'lower-bound ; slot name
   #'(lambda    ; demon function
      (assertion-obj name-of-modified-slot
       old-value new-value)
      (gauge-update assertion-gauge)))
  (add-slot-if-modified-demon
   participant  ; object
   'upper-bound ; slot name
   #'(lambda    ; demon function
      (assertion-obj name-of-modified-slot
       old-value new-value)
      (gauge-update assertion-gauge)))
  assertion-gauge))
(defmethod application-visualization-coupler
 ((participant or-box))
 (make-label "OR"))
```

Demon functions are closures which provide access to the corresponding visualization object. The gauges for assertion objects (broker, etc.) use indirect values to access assertion objects. The objects representing labels for constraints are the same as in the class browser example.

6.3 Method Demons

Slot demons offer an elegant way of defining slot accesses as interesting events and hence updating corresponding visualization objects. Not only slot accesses are subject to updating a visualization. Every method might define an event of interest. Slot accesses are only special cases. General *method demons* can be implemented using the meta-object protocol of CLOS. The idea[5] is to wrap a method with a so-called *wrapper method* which has slots to refer to both the demon functions and the original method (see Figure 6). When all demons are removed the wrapper method itself is removed, too. In this case there is no overhead as with a metaclass which provides own methods for slots accesses that overwrite the standard slot accessor methods (e.g. for indirect values).

A major disadvantage of this wrapping slot accessor is that demons are evaluated for all instances, i.e. they are slot

[4] We refer to [12].

[5] An implementation proposal for CLOS was originally outlined by Gregor Kiczales.

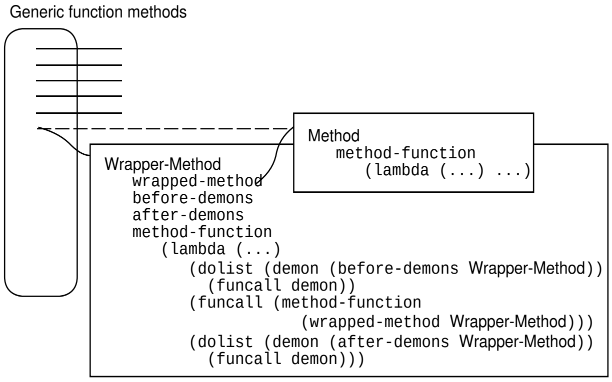

Figure 6: Outline of a wrapper method.

but not instance-specific. Method demons do not solve the problem of compound slot accesses, either.

6.4 Instance-Specific Meta-Objects

The CLOS meta-object system assigns to metaclasses the responsibility for both structure (implementation) and behavior of instances. There are other meta-level systems which distinguish between structural and computational meta-objects [10]. In this section we present ideas to provide some kind of dynamic meta-level influence in CLOS [7]. We implement meta-objects as instances of the class `standard-meta-object` which is not a CLOS metaclass. The standard slot access protocol which uses the method `slot-value-using-class` is analogously extended for these "simple" meta-objects.

```
(defclass standard-meta-object () ())
(defmethod slot-value-using-meta-object
 ((mobj standard-meta-object) object slot-name)
 (call-next-meta-method))
(defmethod (setf slot-value-using-meta-object)
 ((mobj standard-meta-object) object slot-name)
 (call-next-meta-method))
```

Meta-objects can be assigned to instances with metaclass `extensible-standard-class`. This metaclass describes classes with instances that have one additional or implicit slot called `meta-objects` (see Figure 7). A set of meta-objects can be assigned to this slot.

An example method handling slot accesses is defined as follows.

```
(defmethod slot-value-using-class
 ((class extensible-standard-class) object slot-name)
 (if (eq slot-name 'meta-objects)
  (call-next-method)
  (let
   ((*meta-objects* (slot-value object 'meta-objects))
    (*meta-class-generic-function*
     #'(lambda () (call-next-method)))
    (*meta-object-generic-function*
     #'(lambda (meta-object)
        (slot-value-using-meta-object meta-object
                                      object
                                      slot-name))))
```

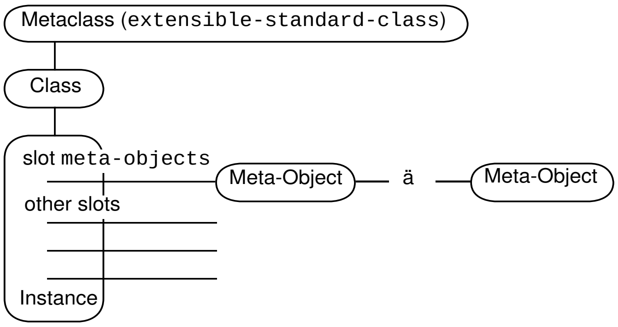

Figure 7: Meta-objects for instances with metaclass `extensible-standard-class`.

```
   (declare (special *meta-objects*
                     *meta-object-generic-function*
                     *meta-class-generic-function*))
   (if (null *meta-objects*)
    (call-next-method)
    (call-next-meta-method)))))
```

The function `call-next-meta-method` is comparable to the predefined function `call-next-method`. It evaluates `slot-value-using-meta-object` for the "next" meta-object in the list of meta-objects (see Figure 7). The default behavior of `slot-value-using-meta-object` is to evaluate `call-next-meta-method` again (s.a.). This default behavior may be augmented or overwritten by subclasses of `standard-meta-object` (s.b.). When there are no meta-objects (left), `call-next-meta-method` invokes the "normal" slot access functionality of `standard-class`. There is also some additional code needed to enable passing of different parameters to the next metamethod just as with `call-next-method`.

```
(defun call-next-meta-method ()
 (declare (special *meta-objects*
                   *meta-object-generic-function*
                   *meta-class-generic-function*))
 (if (endp *meta-objects*)
  (funcall *meta-class-generic-function*)
  (funcall *meta-object-generic-function*
           (pop *meta-objects*))))
```

We use these meta-level techniques to extend our constraint example. Using the protocol described above visualizations of particular instances can be provided with little programming effort. For instance, a meta-object could be defined by the class `visualizer-meta-object`. This class combines a visualization object with a list of interesting slots. Every writing access to these slots is followed by calling the instance-specific visualization object.

```
(defclass visualizer-meta-object
 (standard-meta-object)
 ((visualizer :initarg :visualizer
              :accessor visualizer
              :initform #'(lambda (&rest ignore) nil))
  (interesting-slots :initarg :interesting-slots
                     :reader interesting-slots))
 (:default-initargs :interesting-slots nil))
```

```
(defmethod (setf slot-value-using-meta-object)
 :after (new-value (mobj visualizer-meta-object)
         object slot-name)
 (if (member slot-name (interesting-slots mobj))
  (funcall (visualizer mobj) object slot-name)))
```

We define a new metaclass for assertion objects which combines slot demons and meta-objects.

```
(defclass extensible-standard-class-with-slot-demons
 (extensible-standard-class demon-slots-class) ())
```

Be `your-opinion` the assertion object of our constraint example. We add only to this object a corresponding meta-object which prints `your-opinion`'s decision about buying stocks. This behavior can be easily reverted by removing this meta-object from the implicit slot `meta-objects` (see Figure 7).

```
(add-meta-object
 your-opinion
 (make-instance
  'visualizer-meta-object
  :interesting-slots '(lower-bound upper-bound)
  :visualizer
   #'(lambda (assertion slot-name)
      (if (> (assertion-lower-bound assertion) 0.75)
       (print 'buy) ; or any other visual feedback
       (print 'donot-buy)))))
```

Another behavior might be to temporarily modify a reading access to a slot value. After adding a meta-object of class `buying-indicator-meta-object` to the object `your-opinion` each reading access to the slot `lower-bound` of `your-opinion` returns the slot value and a buying indicator.

```
(defclass buying-indicator-meta-object
 (standard-meta-object) ())
(defmethod slot-value-using-meta-object
 ((mobj buying-indicator-meta-object)
  object slot-name)
 (if (eq slot-name 'lower-bound)
  (let ((slot-value (call-next-meta-method)))
   (if (> slot-value 0.75)
    (values slot-value 'buy)
    (values slot-value 'donot-buy)))
  (call-next-meta-method)))
(add-meta-object
 your-opinion
 (make-instance 'buying-indicator-meta-object))
```

One may of course argue that this implementation is a little impure because of using different mechanisms: metaclasses and meta-objects. Moreover not all meta-objects may be compatible. There remains also some overhead although no meta-objects are attached to an instance.

7 Related Work

The Symbolics™ programming environment Genera™ offers also means for specifying layout of windows [26]. Subwindows (panes) can be arranged in a frame in columns or rows. Window sizes can be determined as absolute (fixed), relative, or with respect to objects being allocated. These features can be compared with our box model. Filler specifications are also supported. Size specifications can be constrained by minimal and maximal distances. Genera only supports layouts for panes, but our layout algorithms can be applied to every object conforming to the underlying abstract protocol. Genera offers the notion of presentation types which can be compared with our view item classes. Presentation types are also associated with handling user input. In contrast to Genera our approach offers a more uniform and orthogonal layout scheme combined with a compact and elegant TEX-notation.

Recently, two other approaches were proposed which use TEX-like layout schemes for user interfaces. They also use constructs such as boxes and fillers for expressing window layout. The InterViews System [21] is a user-interface toolkit based on X windows and implemented in C^{++}. Fillers and boxes are implemented as objects. With respect to our Lisp environment we found the representation of boxes as a combination of ordinary lists and macros more efficient for manipulation and pattern matching. Our layout scheme is in several respects more powerful than InterViews' scheme. A filler-like size specification of boxes is not possible in InterViews. Important and useful notions such as a frame box which constrains the size of a its box element or a general box which invokes user-defined parsers for layout specifications are not available.

The FormsVBT system [1] offers a two-view approach to designing user interfaces. The layout of a dialog window can be specified using both a TEX-like textual and a direct-manipulative graphical representation. Changes made in either representation are immediately updated in the other representation. FormsVBT is implemented in a dialect of Modula-2. Its specification language supports no macros and offers no support for new box types and layout schemes. Furthermore, we see the problem that the functionality of the textual specification notation has to conform with the graphical user interface. Mostly, this requires to reduce the functionality of the textual notation.

8 Summary and Future Work

This paper presented a framework for visualizing object-oriented systems. It consists of a compact, flexible notation for specifying layout of graphical objects. This notation is fully integrated into a Lisp environment based on CLOS. Advantages of this TEX-like notation are its expressiveness, user-predictable layouts, and efficient implementation schemes. The CLOS meta-level architecture is used to associate visualization and application objects. Supported techniques are indirect values, slot and method demons, and instance-specific meta-objects. These visualization techniques require no modifications to the systems which are selected for visualization.

Next steps might be to combine the advantages of TEX-style notations with the general flexibility of constraint systems. Another useful extension to box specifications might be to support local variables which could represent box attributes such as box width and height (see [14] for details). We also plan to address the problem of interpreting several different generic functions as a single interesting event. One solution might be to define higher-level demons combining demons of different methods. Furthermore, research is necessary to extend this approach to 2-1/2 or 3-D layout.

Acknowledgements

We like to thank Roman Cunis and Ken Kahn for comments on a draft of this paper. The first author has been partly supported by a DAAD scholarship granted by the NATO science committee.

References

[1] G. Avrahami, K.P. Brooks, M.H. Brown, A Two-View Approach to Constructing User Interfaces, *ACM Computer Graphics* **23**, 3 (July 1989), 137–146.

[2] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon, Common Lisp Object System Specification, *ACM Sigplan Notices* **23**, 9 (Sept. 1988).

[3] D.G. Bobrow, M. Stefik, The LOOPS Manual, Xerox Corporation, December 1983.

[4] A. Borning, The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems* **3** (1981), 353–387.

[5] M.H. Brown, Algorithm Animation, ACM Distinguished Dissertations Series, MIT Press, 1988.

[6] P.T. Cox, F.R. Giles, T. Pietrzykowski, Prograph: a step towards liberating programming from textual conditioning, In: Proceedings, *1989 IEEE Workshop on Visual Languages,* Rome (Italy), Oct. 4-6, IEEE Computer Society Press, 1989, pp. 150–156.

[7] R. Cunis, Some Ideas on Integrating Reflective Aspects into CLOS-type Object Systems, Internal Report, University of Hamburg, Computer Science Department, 1990.

[8] W. Cunningham, K. Beck, A Diagram for Object-Oriented Programs, *ACM Sigplan Notices* **21**, 11 (1986), 361–367.

[9] M. Eisenstadt, M. Brayshaw, The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming, *Journal of Logical Programming* **5** (1988), 277–342.

[10] J. Ferber, Computational Reflection in Class-Based Object-Oriented Languages, *ACM Sigplan Notices* **24**, 10 (1989), 317–335.

[11] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Mass., 1983.

[12] N. Graube, Metaclass Compatibility, *ACM Sigplan Notices* **24**, 10 (1989), 305–315.

[13] V. Haarslev, R. Möller, VIPEX: Visual Programming of Experimental Systems, In: *Visual Languages and Visual Programming,* S.K. Chang (ed.), Plenum Press, New York and London, 1990, pp. 185–212.

[14] V. Haarslev, R. Möller, Visualization and Graphical Layout in Object-Oriented Systems, Technical Report, System Sciences Lab, Xerox PARC, 1990.

[15] V. Haarslev, R. Möller, A Declarative Formalism for Specifying Graphical Layout, In: [28], pp. 54–59.

[16] K.M. Kahn, V.A. Saraswat, Complete Visualizations of Concurrent Programs and their Executions, In: [28], pp. 7–14. See also *Technical Report* SSL-90-38 [P90-00099], Xerox Palo Alto Research Center, 1990.

[17] S. Keene, Object-Oriented Programming in CLOS - A Programmer's Guide to CLOS, Addison-Wesley, 1989.

[18] K. Kimbrough, O. LaMotte, Common Lisp User Interface Environment, Preprint, Texas Instruments Inc., July 1989.

[19] M.F. Kleyn, P.C. Gingrich, GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views, *ACM Sigplan Notices* **23**, 11 (1988), 191–205.

[20] D.E. Knuth, $\TeX$ and Metafont – New Directions in Typesetting, Digital Press, 1979.

[21] M.A. Linton, J.M. Vlissides, P.R. Calder, Composing User Interfaces with InterViews, *IEEE Computer* **22**, 2 (1989), 8–22.

[22] J.H. Maloney, A. Borning, B.N. Freeman-Benson, Constraint Technology for User-Interface Construction in ThingLab II, *ACM Sigplan Notices* **24**, 10 (1989), 381–388.

[23] S. Matwin, T. Pietrzykowski, PROGRAPH: A Preliminary Report, *Computer Languages* **10**, 2 (1985), 91–126.

[24] R. Möller, AI-Based Visualization Tools in Object-Oriented Systems (in German), *Technical Report* FBI-HH-B-149/90, University of Hamburg, Computer Science Department, 1990.

[25] R.E. Roberts, I.P. Goldstein, The FRL Manual, AI Memo 409 Edition, MIT Lab., 1977.

[26] Handbooks of Symbolics Programming Environment, 7A, Programming the User Interface – Concepts, Symbolics Inc., 1988.

[27] P.A. Szekely, B.A. Myers, A User Interface Toolkit Based on Graphical Objects and Constraints, *ACM Sigplan Notices* **24**, 10 (1989), 36–45.

[28] Proceedings, *1990 IEEE Workshop on Visual Languages,* Skokie, Illinois, Oct. 4-6, IEEE Computer Society Press, 1990.

[29] P.H. Winston, B.K.P. Horn, LISP, 3rd edition, Addison-Wesley, 1989.

[30] S. Wright, W. Feuerzeig, J. Richards, pluribus: A Visual Programming Environment for Education and Research, in: 1988 IEEE Workshop on Languages for Automation, Symbiotic and Intelligent Robotics, Univ. of Maryland, College Park, Maryland, Aug. 29–31, IEEE Soc. Press, 1985, pp. 29–31.