THE PICASSO APPLICATION FRAMEWORK

by

Lawrence A. Rowe, Joe Konstan, Brian Smith
Steve Seitz, and Chung Liu

Memorandum No. UCB/ERL M90/18

14 March 1990
(Revised 23 May 1991)

THE PICASSO APPLICATION FRAMEWORK

by

Lawrence A. Rowe, Joe Konstan, Brian Smith,
Steve Seitz, and Chung Liu

Memorandum No. UCB/ERL M90/18

14 March 1990
(Revised 23 May 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

THE PICASSO APPLICATION FRAMEWORK

by

Lawrence A. Rowe, Joe Konstan, Brian Smith,
Steve Seitz, and Chung Liu

Memorandum No. UCB/ERL M90/18

14 March 1990
(Revised 23 May 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The PICASSO Application Framework†

Lawrence A. Rowe, Joe Konstan, Brian Smith, Steve Seitz, and Chung Liu

Computer Science Division-EECS
University of California
Berkeley, CA 94720

Abstract

PICASSO is a graphical user interface development system that includes an interface toolkit and an application framework. The application framework provides high-level abstractions including modal dialog boxes and non-modal frames and panels similar to conventional programming language procedures and co-routines. These abstractions can be used to define objects that have local variables and that can be called with parameters.
PICASSO also has a constraint system that is used to bind program variables to widgets, to implement triggered behaviors, and to implement multiple views of data. The system is implemented in Common Lisp using the Common Lisp Object System and the CLX interface to the X Window System.

Keywords: Graphical User Interface Development Environment, Application Framework, User Interface Toolkit, User Interfaces

1. Introduction

PICASSO is a graphical user interface development system that includes an interface toolkit and an application framework. The toolkit contains a library of predefined interface abstractions (e.g., buttons, scrollbars, menus, forms, etc.), geometry managers, and a constraint system. The application framework provides high-level abstractions and other infrastructure to support the development of graphical user interface applications.

The PICASSO framework includes five object types: applications, forms, frames, dialog boxes, and panels. An *application* is composed of a collection of frames, dialog boxes, and panels. A *form* contains fields through which data can be displayed, entered, or edited by the user. A *frame* specifies the primary application interface. It contains a form and a menu of operations the user can execute. A *dialog box* is a modal interface that solicits additional arguments for an operation or user confirmation before executing a possibly dangerous operation (e.g., deleting a file). A *panel* is a nonmodal dialog box that typically presents an alternative view of data in a frame or another panel.

†This research was supported by the National Science Foundation under grants DCR-85-07256 and MIP-87-15557. The second author was supported by a NDSEG fellowship administered by DARPA.

Figure 1 shows a screen dump of a sample application that displays information about employees and departments. The frame, shown on the left, displays information about an employee. It contains a form with fields that describe the employee (e.g., name, age, etc.). Above the form is a menu-bar with pull-down menus that contain operations the user can execute. The buttons at the bottom of the frame allow the user to step through the employees in the database. The panel on the right displays information about the department to which the employee belongs. At the top of the panel is a hierarchy browser that lists departments and the employees in a selected department. Information about the department that the current employee belongs to is shown below the browser. The department information includes the manager and a graphics field that shows a floor plan with the selected employee and his or her manager's offices highlighted. If the user selects a button at the bottom of the frame to display the previous or next employee and that employee is in a different department, the department information in the panel is automatically changed (i.e., the data displayed through the frame and panel are synchronized).

PICASSO is an object-oriented system implemented in Common Lisp that runs on the X Window System [21]. The toolkit, framework, and user applications are implemented as Common Lisp Object System (CLOS) objects [7]. A CLOS class is defined for each type of framework object (e.g., application, frame, form, dialog

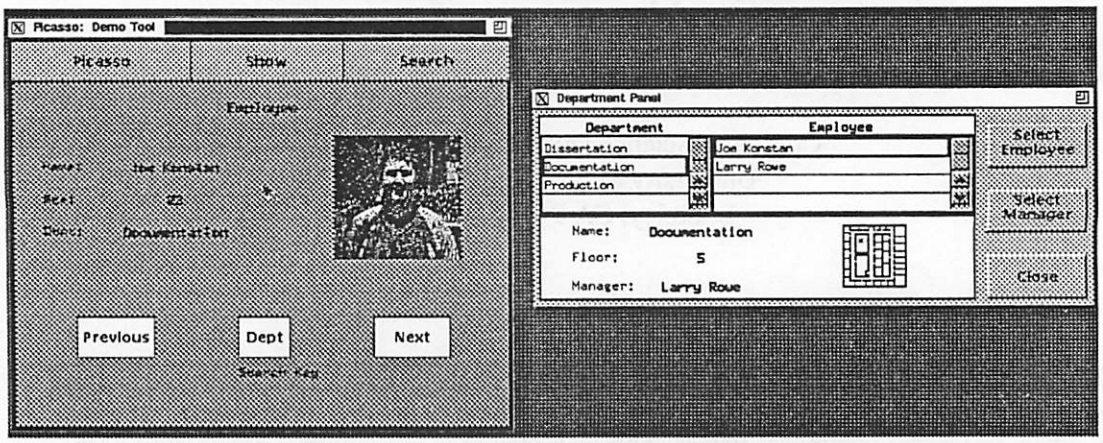


Figure 1: Example PICASSO application interface

box, and panel). Instances of these classes are called PICASSO objects (PO's). Each PO type has a different visualization and control regime. The toolkit widgets that implement the visualization and control (e.g., title bars, buttons and menus) are automatically generated when a PO is created.

PO's are similar to procedures and functions in a conventional programming language. They have a name, local variables, formal arguments, and a lexical parent. A PO can be called and arguments passed to it which causes the PO to allocate space for its local variables and to create X resources to display the values of selected variables. The user can examine and edit the data or execute code attached to buttons and menu operations. Code can be arbitrary Lisp expressions that can, for example, change the values of variables, call other PO's, return to the calling PO, or call a Lisp function.

PO's are analogous to familiar programming language concepts (e.g., procedures, functions, and co-routines). Frames are similar to procedures. Only one frame can be active at a time. Calling a frame conceals the current frame and displays the new frame. Returning from the called frame redisplays the calling frame. Dialog boxes are similar to functions. Calling a dialog box displays it and the user is forced to respond. A dialog box returns a value to the caller when it returns (e.g., "OK"). Panels are similar to co-routines. Calling a panel displays it in a separate window and the user can interact with it or any other frame or panel. The location of the mouse cursor determines which frame or panel receives user inputs.

Variables can be passed to a PO as parameters in the call. For example, the current employee in the frame above is passed to the panel by reference so that the value can be changed in either the frame or panel and the update will be propagated to the other PO. Traditional *value*, *value-result*, and *reference* parameter passing are provided. In addition, two new mechanisms are provided that implement different kinds of synchronization. *Value/update* parameters are similar to *value* parameters except that subsequent updates to the actual argument are propagated to the copy in the called PO. *Value-result/update* is similar to *value/update* except the value is copied back to the calling environment when the PO returns. These parameter passing mechanisms can be used when changes to a value displayed through a panel should not be immediately propagated back to the caller, but changes to the actual argument should be propagated into the panel.

PO's are stored in an external database and loaded into the application when needed. They are shared by different applications because the database is shared. Commonly used PO's (e.g., a file directory browser and an error message dialog box) are provided to maintain interface consistency between different applications.

PICASSO provides a collection of widgets (as is provided by toolkits such as Motif[19] or Open Look[6]) including color graphics and tables. PICASSO also provides capabilities that are similar to other application frameworks including Garnet [17], InterViews [13], MacApp [23], NextStep[18], and Smalltalk [5]. PICASSO differs from these frameworks in that it clearly separates the framework layer from the toolkit layer. The framework provides support for defining and accessing local variables, binding variables to fields in a form, and passing parameters to other PO's. These mechanisms encourage the development of reusable interface components. It is used to specify bindings between variables and different

views (i.e., dialog boxes, frames, and panels). Parameter passing is implemented by a constraint system that is similar to the constraint systems in Grow [1], ThingLab II [14], and Garnet. The constraint system is also used to specify interface abstractions (e.g., scrollbars) as in Garnet.

A PICASSO frame is similar to the Smalltalk Model-View-Controller (MVC) abstraction [11]. A frame contains local variables which correspond to the model. The form in a frame is the view. And, a frame implicitly defines the controller. Multiple views of the same data can be created by calling a panel and passing the appropriate local variable. The way a variable is passed determines whether or not and when view updates are propagated back to the frame. The Navigator Framework extension to MVC developed at ParcPlace Systems provides a similar capability.

We believe that using conventional programming language abstractions simplifies the problem of designing and implementing direct manipulation interfaces. In addition, we believe the PICASSO framework is easier to learn and use than the Smalltalk MVC and Navigator abstractions because the application developer can use familiar programming abstractions (e.g., variables and parameter passing) rather than unfamiliar abstractions (e.g., active values, message passing, and MVC).

This paper describes the PICASSO framework and programming model. The programming constructs described below are shown as extensions to Lisp. Most users will not see these textual specifications because a direct manipulation interface builder is being developed to create and modify applications. The interface builder can be used to define any PO. Forms are defined by selecting widgets from a palette and placing them at the desired location in a window with the mouse. Field attributes (e.g., border, default values, etc.) can be changed interactively. Similar interfaces are provided to define other PO types and code. The toolkit and interface builder are extensible so that developers can add new interface abstractions to the system. The toolkit and interface builder are described elsewhere [10].

The remainder of the paper is organized as follows. Section 2 describes an example application used to illustrate features of the application framework. Section 3 describes the framework model and sections 4 through 6 describe the different types of PO's. Section 7 describes the programming constructs used to implement operations. Section 8 discusses the implementation of PICASSO and describes applications written with it. Lastly, section 9 concludes the paper.

2. An Example Application

The personnel browser shown in figure 1 will be used to illustrate how an application is defined using PICASSO. Figure 2 shows the initial frame in the application. The user can scroll through the employees by using the **Previous** and **Next** buttons in the form.

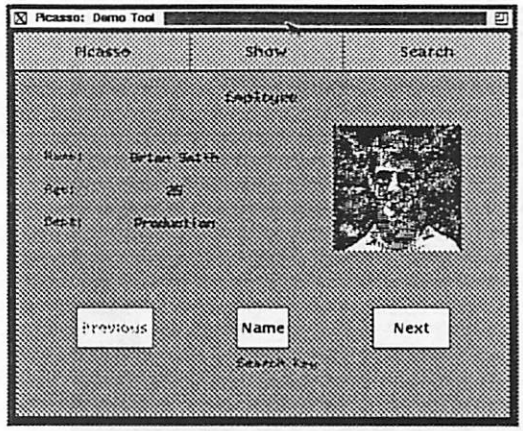


Figure 2: Application Window

The **Search** pull-down menu shown in figure 3 contains operations that let the user search for an employee on any attribute (e.g., age or department). For example, selecting the **By Age...** menu operation calls the dialog box shown in figure 4. The user enters the desired age in the type-in field and presses the **OK** button which returns from the dialog box and changes the display to the employee closest in age to the value entered. The **Previous** and **Next** buttons now search forwards and backwards by age. This search order can also be changed by using the pop-up menu as shown in figure 5.

Department information can be displayed by selecting the **Department...** operation in the **Show** menu which calls the department panel shown in figure 1. This panel contains a hierarchy browser, textual information, and a floor plan of the department. If the user selects a

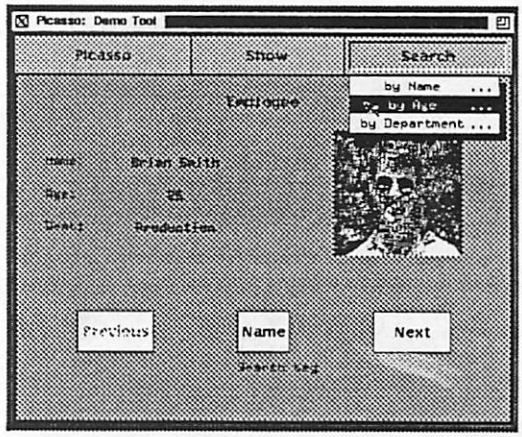


Figure 3: Search Menu

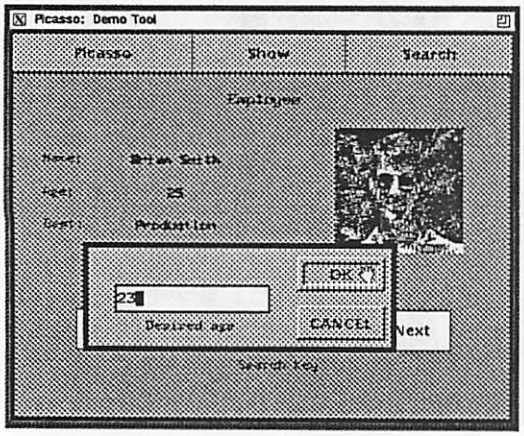


Figure 4: Search Dialog Box

department in the hierarchy browser, the employees in that department are displayed. The floor plan highlights the offices of the current employee and his or her manager. The buttons on the right allow the user to change the current employee either to the selected employee in the hierarchy browser or the current employee's manager or to close the panel. Changing the current employee causes the data in the frame and the panel to be changed.

3. The PICASSO Programming Model

A PICASSO application, also known as a *tool*, is composed of a collection of frames, dialog boxes, and panels. This section describes these PO's and provides an overview of the programming constructs used to implement the semantics of buttons and menu operations.

A form is an input/output abstraction that corresponds to a paper form. It contains fields through which data can be displayed and edited by the user. Data can be displayed in a field using text in different fonts, images, and drawings. Each form maintains a field visit order

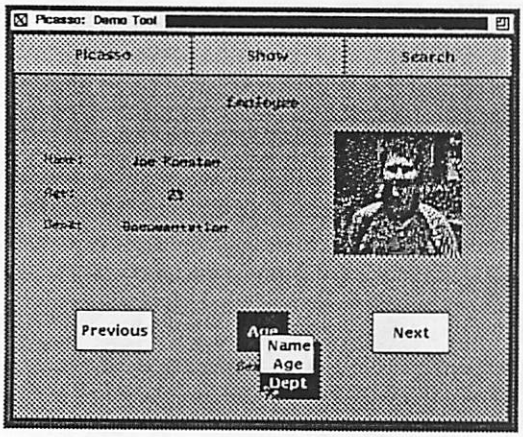


Figure 5: Search Order Menu

and current field and manages the keyboard focus. When the user moves the mouse into the form, the keyboard is focused to the current field. The user can change the current field by selecting another field with the mouse or by using keyboard commands to move to the next or previous field.

Forms also contain display-only labels and decorative trim (e.g., lines and backgrounds). Fields are implemented by widgets and trim is implemented by gadgets in the toolkit. Widgets are input/output abstractions and gadgets are output-only abstractions. In addition, a geometry manager for the form must be specified. The geometry manager is responsible for repositioning the fields and labels when a form is resized.

Forms can be used in frames, panels, and dialog boxes. Alternatively, an implicitly defined form can be used instead by specifying the widgets and gadgets and a geometry manager. The widgets and gadgets are called *children*. The term widget is used below to refer to a field, label, or trim in a form.

A frame is a control abstraction that implements a major application mode or operation. It contains a form and operations the user can execute. Pull-down menus at the top of the frame hold the operations which can perform computations, change values displayed through the form, and call dialog boxes, panels, or other frames. An application may contain many frames, but only one is active at any time.

A dialog box is a control abstraction that requires an immediate response from the user. It requests additional information needed to complete an operation, or it notifies the user about an error that has occurred or might occur. The dialog box is displayed in a separate window that is positioned on top of the PO that called it. Moreover, the user is forced to enter data into it. A dialog box contains a form and a list of operations that are displayed in a column of buttons down the right side of the dialog box.

A panel is a non-modal control abstraction that is used to view or edit data. It contains a form and a list of operations that can be displayed either in a menu-bar at the top of the panel like a frame or in a column of buttons down the right side like a dialog box. The data displayed through the form is typically displayed in a different way in the frame or another panel. For example, the department information displayed in the panel in the sample application also displays information about the current employee. Many panels may be active at once. The location of the mouse cursor determines whether one of the panels or the frame receives input. Several copies of the same panel can be active at the same time. For example, a word processor might use a panel to allow the user to edit style properties of a block of text (e.g., font, typeface, etc.). The same panel can be called with different

text blocks so that the user can edit style properties of the blocks at the same time.

Static and dynamic variables can be declared in any PO. These variables hold data displayed to the user through a form as well as other internal data used by the application. Static variables persist between calls to a PO. They are allocated when the PO is created. Dynamic variables are allocated each time a PO is called and deallocated when it returns.[1] Named constants can also be declared.

PICASSO uses lexical scoping. The lexical parent of a PO is the object in which it is declared. A variable reference that cannot be resolved locally is searched for in the parent. For example, if a variable is referenced in a form, but not defined there, the parent is searched for the variable, followed by the parent's parent and so forth until the application-wide level is reached. If the variable is not found in that level, an error is signaled.

Variables can be used in left- or right-value contexts by using the Lisp function `value`. For example, given PICASSO variables *x* and *y*, the following code assigns *y* to *x*:

```
(setf (value x) (value y))
```

Since variables are lexically scoped, common operation take variable names and use the appropriate value as either a left- or right-value. A Lisp reader macro #! is provided to simplify the code. This rmacro takes a variable name, finds the variable, and provides a left- or right-value reference to that variable.The previous example is normally specified by

```
(setf #!x-name #!y-name)
```

Another reader macro (`#?`) is provided that returns the address of a PICASSO variable. This macro is needed so that PICASSO variables can be passed by reference. The use of reference variables is described below. Other uses of these reader macros are described in section 7.2.

Parameter passing is used to pass values to other PO's when they are called. A dynamic variable is created for each formal parameter. The parameter passing mechanisms provided include: value, value/update, value-result, value-result/update, and reference. Value/update and value-result/update parameters are useful when displaying different views of a data structure through a panel, frame, or dialog box. A PO that implements an editable view is passed the data by reference so that changes are immediately propagated to the caller. A PO that implements a non-editable view is passed the data by value or value/update depending on whether changes made to the data through other views should be propagated to this view. Updates to data passed by value-result or value-result/update are deferred until the user executes an operation that returns from the PO.

The design of PICASSO encourages the development of reusable interface abstractions. We expect that general-purpose panels and dialog boxes (e.g., table browsers and prompters) will be developed and reused in many applications. Similarly, forms can be reused in different panels, dialog boxes, and frames. They can also be reused in other forms. For example, a standard name and address block for a person can be reused in any form that displays information about a person. To encourage reuse, each PO has a unique external name. This name is composed of three parts each of which is a Lisp string:

(package name . suffix)

A package is a collection of related PO's. The optional suffix is used to identify the type of the PO. External names can be used to specify a PO stored in the database or already loaded into main memory.

Most PO's are referenced by name in the definition of their lexical parent which automatically loads it when the parent is called. A shorter internal name can be specified for the PO to simplify the code. In addition, internal names facilitate changing to a different PO since only the internal name binding has to be changed. A function is provided that allows an application to load a PO at runtime.

Procedural code is used to specify the semantics of button presses, menu operations, or setup, initialization, or termination of a PO. In the current implementation, this code is specified in Lisp. It can reference PICASSO or Lisp variables and call PO's or Lisp functions or macros.

4. Applications

An application window is the outermost object in an application. This window is managed by a window manager. Iconifying this window causes all children to be concealed.

Each application maintains a list of packages that are searched when looking for partially-specified PO's. For example, the package search list in the example application contains "`paper`" and "`picasso`". The "`paper`" package contains the PO's that define the application. The "`picasso`" package is automatically included in all search lists by the system. It contains built-in PO's such as dialog boxes to prompt for a file name and to confirm a destructive operation used by all PICASSO applications.

[1] Actually, dynamic variables are not deallocated when a PO returns because it takes a long time to re-establish bindings defined on them. It is more efficient to reuse the same variable on subsequent calls after re-initializing it.

```
(deftool ("paper" "demo" . "tool")
    "Employee-department application"
    (title "Demo Tool")
    (frames (main-frame ("demo" . "frame")))
    (init-code (open-database))
    (exit-code (close-database)))
```

Figure 6: Application Definition

The sample application was defined by the `deftool` call shown in figure 6. The first line specifies the external name and formal arguments for the application. The second line is a documentation string. The remaining lines specify the title, frames, and initialization and exit code. The title is displayed in the application window's title bar. The `frames` clause specifies the frames used in this application. This application has only one frame, named ("demo" . "frame"), that is bound to the internal name `main-frame`. The frame can be found in the "paper" package so the external name does not need to be fully qualified. The `init-code` and `exit-code` clauses specify the code to be executed when the application is run and when it is exited. In this case, these clauses open and close the database.

5. Forms

Forms are used in frames, panels, or dialog boxes. A form that can be reused in more than one PO, called a *pluggable form*, is defined using the `defform` construct. Pluggable forms typically have local variables and parameters and like any PO, they may have initialization and termination clauses that specify code to be executed when the PO that holds the form is called and exited, respectively.

The form used to browse employee information in the sample application was written as a pluggable form. The definition is shown in figure 7. The form name is `("paper" "employee" . "form")`. The clauses in the form definition specify the children, geometry manager, and setup code. The `children` clause specifies the fields and labels in the form. Each "`make-`" call instantiates a widget which can be a predefined widget or gadget in the PICASSO toolkit or a new widget or gadget defined by the user. A field or label specification is preceded by a symbol in some cases to declare a variable bound to that widget. For example, `dept-field` is bound to the text gadget that displays the name of the employee's department. A gadget is used rather than a widget because the application does not allow the user to change the department.

The buttons at the bottom of the form (e.g., **Previous**, **Name**, and **Next**) are also specified. They are combined into a collection gadget so they will be arranged horizontally.

The `gm` clause specifies the geometry manager which lays out the widgets in the form. Many geometry managers are provided by the PICASSO toolkit. Each takes a set of widgets and optional layout parameters, called *geometry specifications*, and calculates an xy-offset for each widget within its parent.

```
(defform ("paper" "employee" . "form") ()
  "This form displays an employee record"
  (gm 'packed-gm)
  (children
    (make-gadget      :value "Employee"
                      :geom-spec '(:top 50)
                      :font "gallant.r.19")
    (make-null-gadget :geom-spec '(:top 20))
    (make-collection-gadget
      :gm 'rubber-sheet-gm
      :geom-spec '(:top 100)
      :children
        '((picture-gadget
           (make-image-gadget
             :geom-spec '(.55 0 .45 .99)))
          (name-field (make-text-gadget :label "Name:"
                        :geom-spec '(.15 0 .35 .33)))
          (age-field  (make-text-gadget :label "Age:"
                        :geom-spec '(.15 .33 .35 .34)))
          (dept-field (make-text-gadget :label "Dept:"
                        :geom-spec '(.15 .67 .35 .33)))
      (make-collection-gadget
      :geom-spec :fill :gm 'rubber-sheet-gm
      :children
        '((pb (make-button  :value "Previous"
              :release-func '(get-emp :dir :prev)
              :geom-spec '(0 0 .5 .47 :center)))
          (kb (make-pop-button :label "Search Key"
              :label-type :botton
              :items '("Name" "Age" "Dept")"
              :geom-spec '(.4 0 .2 1 :center)))
          (nb (make-button   :value "Next"
              :release-func '(get-emp :dir :next)
              :geom-spec '(0 .52 .5 .47 :center)))
  (setup-code
    (progn (bind (value #!name-field) (name #!employee))
           (bind (value #!age-field) (age #!employee))
           (bind (value #!picture-gadget ) #!picture)
           (bind (value #!dept-field) #!dname)
           (blet (dimmed #!pb)
             :var ((e #!employee) (k #!key))
             (not (prev-exists e k)))
           (blet (dimmed #!nb)
             :var ((e #!employee) (k #!key))
             (not (next-exists e k)))
           (blet (value #!kb) :var ((k #!key))
             (symbol-name k))
           (blet #!key :var ((kbv (value #!kb)))
             (make-keyword kbv)))))
```

Figure 7: Form Definition

The `setup-code` clause specifies code that is to be executed when the PO is created. In this case, the code establishes bindings between the variables that hold the data and fields in the form through which it is displayed. Bindings are also established for the buttons at the bottom of the form that will dim **Previous** and **Next** if records do not exist and that reset the variable `#!key` to hold the search attribute. Note that the variable `#!key` is declared in the frame, and it is accessed through lexical scoping. The functions used to define the bindings are described in section 7.3.

Sometimes forms are only used in a single frame, panel, or dialog box. It complicates the application specification if the developer has to create a separately named form so the developer can specify the children and other form clauses directly in the frame, panel or dialog box specification. These forms are called *implicit forms* and they cannot have local variables or parameters. They can, however, access variables and parameters in their lexical parent. Examples of implicit forms are presented below.

6. Frames, Dialog Boxes, and Panels

Frames, dialog boxes, and panels are callable PO's. They are typically called in response to a user action (e.g., a menu selection or button press) as follows:

(call *PO* :arg-1 *value* :arg-2 *value* ...)

The PO is specified by an expression that evaluates to a pointer that references the appropriate PICASSO object. The expression is usually the internal PO name. Parameters are passed using Lisp keyword-value pairs.

The semantics of calling a PO are:

(1) Fetch the PO from the database, if it is not already in memory.
(2) Bind the actual arguments to the formal arguments.
(3) Allocate and initialize local variables.
(4) Fetch the lexical children of the PO (e.g., forms, frames, etc.), if they are not already in memory.
(5) Execute the `init-code` for the PO.
(6) Display the object on the screen.
(7) Enter an event loop.

PO's are cached in main memory to avoid the delays inherent in accessing the database. Lexical children are fetched when the PO is called to improve the performance of subsequent calls. Recall that dynamic variables are allocated on each call and static variables are allocated when the PO is created. The event loop dispatches all events (e.g., mouse, keyboard, redraw, etc.) to the appropriate event handlers.

The following code returns from a PO:

(ret *PO optional-return-value*)

This code is executed in response to a user action (e.g., a menu selection or button press) or because a lexical parent is cleaning up its children before exiting. The semantics of returning from a PO are:

(1) Force active lexical children to execute a return.
(2) Execute the `exit-code`.
(3) Conceal the PO which erases it from the screen.
(4) Copy result arguments back to the actual arguments.
(5) Re-enter the event loop of the calling PO.

The remainder of this section, describes how callable PO's are defined.

6.1 Frames

A frame can specify a named form or a set of children widgets through which data will be displayed to the user. Variables defined in the frame, called *frame variables*, store the data on which the frame operates. Forms, panels, and dialog boxes in the frame can access this data by referencing the frame variables or the frame can pass

```
(defframe ("paper" "demo" . "frame") ()
  "This is the only frame of the Paper Demo appplication"
  (static-variables employee department key)
  (panels (dept-panel ("department" . "panel")))
  (dialogs (search-dialog ("search" . "dialog")))
  (form (emp-form ("employee" . "form")))
  (menu-bar
    ("Show" "Show Related Information"
      (dept-entry
        ("Department"
          (progn (call #!dept-panel :emp #?employee
                                :dept #?department)
                 (setf (me-dimmed #!dept-entry) t)))))
    ("Search "Search Employee"
      ("by Name ..."
        (setf #!employee
          (get-emp :name
            (call #!search-dialog :entity "name"))))
      ("by Age ..."
        (setf #!employee
          (get-emp :age
            (call #!search-dialog :entity "age"))))
      ("by Department ..."
        (setf #!employee
          (get-emp :department
            (call #!search-dialog :entity "dept")))))))
  (setup-code
    (progn    (setf #!employee (get-first-emp))
              (blet #!department :var ((e #!employee))
                    (department e)))))
```

Figure 8:Frame Definition

data to them as arguments.

A frame is defined using the `defframe` construct. Figure 8 shows the code that defines the frame in the sample application. Three variables, two menus, a form, a panel, and a dialog box are defined in this frame. The variables `employee` and `department` point to CLOS objects that represent an employee and his or her department. The variable `key` holds the current search key (e.g., age or name).

The menu-bar clause defines two menus: **Show** and **Search**. A menu specification includes: 1) the menu name (e.g., **Search**), 2) a long name that will be displayed at the top of the menu if it is torn off (e.g., **Search Employee**), and 3) a list of operation specifications. An operation specification includes: 1) an optional variable name for the menu entry (e.g., `dept-entry` in the **Department** operation in the **Show** menu), 2) an operation name, 3) the code to execute when the operation is selected, and 4) operation options (e.g., dimmed, left or right string, etc.). The **PICASSO** menu, automatically supplied by the system, is the leftmost menu in the menubar. It contains operations that are useful everywhere (e.g., **Help**, **Print Window**, **Quit**, etc.).

The **Department** operation in the **Show** menu calls the department panel and passes the employee and department objects to it. They are passed by reference so that changes made in the frame will be propagated to the panel and vice versa. The operations in the **Search** menu change the order in which records are scanned.

6.2 Dialog Boxes

Dialog boxes are defined using the `defdialog` construct. The code shown in figure 9 defines the search dialog box in the sample application. The dialog box has a single value parameter, named `entity`, that is set to a Lisp keyword that indicates the type of search being performed (e.g., age or name) and a variable, named `prompt-text`, that is set to a string that prompts the user (e.g., "**Desired age**"). The dialog box has an implicit form so it has `children` and `gm` clauses. Two buttons are specified: **OK** and **Cancel**. The setup code binds `prompt-text` to the type-in field label so an appropriate prompt will be displayed. The rest of the defdialog structure is similar to the other PO definitions.

```
(defdialog ("paper" "search" . "dialog") (entity)
   (dynamic-variables
      (prompt-text (string-concat "Desired " #!entity)))
   (children
      (type-in (make-entry-widget    :value ""
                                     :geom-spec :center)))
   (buttons
      ("OK (ret self (if (eql (value #!type-in) "")
                         nil (value #!type-in))))
      ("Cancel" (ret self nil)))
   (gm 'rubber-sheet-gm)
   (init-code (setf (value #!type-in) nil))
   (setup-code (bind (label #!type-in) #!prompt-text)))
```

Figure 9: Dialog Box Definition

6.3 Panels

Panels are defined using the `defpanel` command. The code shown in figure 10 defines the department panel in the sample application. The panel takes two reference arguments (i.e., `emp` and `dept`) that point to the current employee and the employee's department. Changes made to these variables in the panel are propagated to the variables in the calling frame. Moreover, if the panel is active, changes made to the frame variables

```
(defpanel ("paper" "department" . "panel") (&ref emp dept)
   (title "Department Panel")
   (children
      (bw (make-browse-widget :geom-spec '(:top 100)
               :col-widths '(1 1.5) :data "all-employees"
               :sort-keys '( ("Department" . ,#'dname)
                             ("Employee" . ,#'name))))
      (make-collection-widget :gm 'rubber-sheet-gm
            :geom-spec '(:bottom 0)
            :children
               '((gg (make-image-gadget
                     :geom-spec '(.55 0 .45 .99 :center)))
               (dname-field (make-text-gadget :label "Name:"
                  :geom-spec '(.25 0 .25 .33 :center-y)))
               (floor-field (make-text-gadget :label "Floor:"
                  :geom-spec '(.25 .33 .25 .34 :center-y)))
               (mgr-field (make-text-gadget :label "Manager:"
                  :geom-spec '(.25 .67 .25 .33 :center-y))))))
   (gm 'packed-gm)
   (buttons
      (sel-emp
         ("SelectEmployee"
            (get-emp :name
               (name (car (current-selection #!bw))))))
      (sel-mgr
         ("Select Manager"
            (get-emp :name (mgr #!emp))))
      ("Close (progn (setf (me-dimmed #!dept-entry) nil)
                     (ret self))))
   (setup-code
      (progn (bind (value #!dname-field) (dname #!dept))
               (bind (value #!floor-field) (floor #!dept))
               (bind (value #!mgr-field) (mgr #!dept))
               (bind (value #!gg) (floor-plan #!emp))
               (blet (dimmed #!sel-emp)
                     :var ((sel (current-selection #!bw)))
                     (null sel))
               (bind (dimmed #!sel-mgr) (dimmed #!sel-emp))
               (blet (current-selection #!bw)
                  :var ((d #!dept) (e #!emp))
                  nil))))
```

Figure 10: Panel Definition

are propagated to the panel.

The remainder of the definition specifies the panel buttons (e.g., **Select Employee**, **Select Manager** and **Close**), an implicit form, and bindings between the variables, form widgets, and menu operations.

7. Programming Constructs

This section describes the programming constructs added to Lisp to simplify the development of PICASSO applications. Topics discussed include: procedural code, variables and constants, and bindings.

7.1 Procedural Code

The environment in which procedural code is executed is defined by the PO that contains it (e.g., the frame for menu operations). This environment includes two implicitly defined Lisp variables: `self` and `event`.[2] `Self` is bound to the object that holds the code (e.g., the button, menu pane, or PO) and `event` is bound to a description of the event that caused the code to be executed (e.g., a "button press" event). All PICASSO variable lookups are performed relative to the value of `self`. For example, the code in the Department operation in the Show menu in the frame calls the department panel as follows:

```
(call #!dept-panel    :emp #?employee
                      :dept #?department)
```

This code references three PICASSO variables: `dept-panel`, `employee`, and `department`. The value of `dept-panel` is a pointer to the panel object. The addresses of `employee` and `department` are passed to the panel because both arguments to the panel are reference parameters.[3]

7.2 Variables and Constants

Variables are created automatically when a PO is created or called. All PO definitions can have clauses to define static- or dynamic-variables. Static-variables are created when the PO is created. Different invocations of the PO reference the same variables. Dynamic-variables are created when the PO is called. Different invocations reference different variables.

Static-variables can be created by the application at run-time using the `add-var` function. For example,

(add-var *variable-name place*)

creates a static-variable named *variable-name* in the PO specified by *place*. The variable is immediately visible to lexical children of the PO.

Named constants can be specified with the `constants` clause. They behave just like variables, except the value cannot be changed. Named constants can also be created implicitly in other clauses of a PO definition. For example, all lexical children of a PO (i.e., PO's specified in the `frames`, `forms`, `panels`, or `dialogs` clauses) are given names that are constants in the parent PO. For example, `emp-form` and `dept-panel` are constants in the sample frame.

Widgets specified in the `children` clause of a PO definition can also be bound to named constants by replacing the widget definition

(make-*widget args*)

with a pair

(*constant-name* (make-*widget args*))

This construct creates a name that references the widget when the PO is instantiated. The same technique can be used with buttons specified in panels and dialog boxes and with menus specified in frames or panels.

Recall that ariables and constants are referenced by using the "`#!`" and "`#?`" macros. The value of the variable is looked up in the current environment. As described above, the current environment depends on which PO is active and the location of the mouse cursor. The setup, initialization, and termination code is always executed in the context of the defining PO.

Once the current environment is established, variable lookup proceeds in a lexical fashion. The variables in the PO referenced by `self` are searched first, followed by the PO that is the parent of `self`. Parent links are followed up to the application window. For ease of use, `#!po` always refers to the closest PO. For example, it is the PO itself if the current lexical environment (i.e., `self`) points to a PO. Otherwise, it is the closest enclosing PO. The variable `#!po` can be used in button or menu code to locate the enclosing PO since `self` points to the button or menu entry.

Sometimes it is necessary to specify where to look for a variable. For example, a frame's initialization code might define bindings between frame variables and widgets in the enclosed form. The syntax "`#!`*variable-name*`@`*place*" evaluates *place* to find a starting point for the search for *variable-name*. More complicated search

---

[2] These were implented as Lisp variables instead of PICASSO variables because they should be visible from all lexical scopes and because this implementation still allows the use of PICASSO variables named "self" and "event."

[3] Most languages do not require the program to specify the address of reference parameters at the point of the call. This nonstandard usage was required because our implementation uses the Lisp evaluator to evaluate code.

paths can also be used to reference variables in different environments. For example, the expression

```
#!start-frame@(current-tool)/x
```

references the variable `x` in the `start-frame` in the current application. Any number of "/"-separated names may occur. The "@" clause can only be used on the first variable, since the other names are located based on the value of the preceding expression. Notice that the location specifier in the "@" clause can be any Lisp expression, including a call, in this case, to the function `current-tool`.

Figure 11 shows a partial list of the variables defined in the sample application and how they can be referenced when the current environment is the department panel.

7.3 Bindings

Bindings are used to maintain consistency between variables and values displayed to the user and to declare constraints between variables and values in the application. Any PICASSO variable or CLOS object slot can be bound to other variables, slots, or functions of variables and slots. The simplest case merely binds a single variable or slot to another, for example the setup code for the sample panel includes:

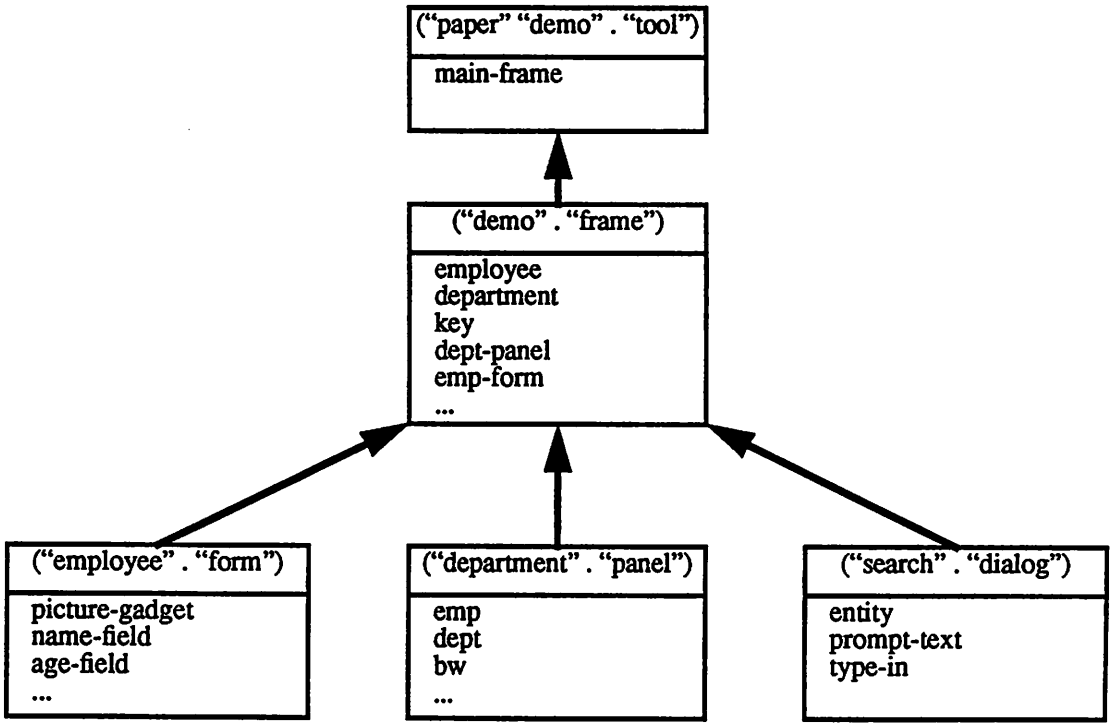
```
(bind (value #!gg) (floor-plan #!emp))
```

This binding declares that whenever the `floor-plan` slot of the CLOS object pointed to by the PICASSO variable `#!emp` changes, the `value` slot of the image gadget pointed to by `#!gg` will be updated to the new value. In this case, whenever the `floor-plan` slot of the structure pointed to by `#!emp` changes, the graphical viewer is updated to display the new floor plan.

The general syntax for the `bind` operation is:

(bind *bound-slot-or-var triggering-slot-or-var*)

A more flexible constraint declaration is available to bind a slot or variable to a function of one or more other slots or variables. The `blet` operation specifies the slots or variables to trigger the change, the slot or variable be-



Expression	PO	Variable
#!emp	panel	emp
#!key	frame	key
#!emp-form/name-field	form	name-field
#!name-field@(find-po-named '("employee" . "form"))	form	name-field

Figure 11:Non-local Variable References

ing bound, and a function to compute the new value. For instance, the setup code in the sample form includes:

```
(blet (dimmed #!pb)
    :var ((e #!employee) (k #!key))
    (not (prev-exists e k)))
```

This code indicates that whenever the `employee` or `key` variables change, the dimmed slot of the **Previous** button should be set on when no previous employee exists and off otherwise. In other words, the button is dimmed whenever there is no previous employee along the specified search key.

A binding is a one-way constraint. Dimming the button would have no effect on the variables. Multiple bindings can be defined to implement a two way constraint. A typical two-way constraint binds a PICASSO variable to a display widget and vice versa. In this way, the display is updated if the variable's value is changed (e.g., in another panel or dialog) and the variable is updated if the user types a new value into the widget. Cycles in bindings are implemented by iterating to a fixed point value. The iteration terminates when the new value assigned to a variable or slot is the same as the current value.

This binding mechanism is relatively simple, but it has proven extremely valuable in implementing user interfaces. The implementation of the binding mechanism is discussed in [8].

8. Discussion

This section describes our experiences implementing the PICASSO framework, some sample applications that have been implemented using it, and the current status of the system.

It has taken approximately ten man-years over a four year period to design and implement PICASSO. During that time, we have implemented:

(1) a foreign function interface to X10 (XCL) [15],
(2) CLOS abstractions for X entities such as displays, windows, fonts, and colors (XCLOS) [16],
(3) a low-level POSTGRES interface (libpq) [26],
(4) a persistent CLOS interface to POSTGRES (SOH) [25],
(5) two INGRES query language interfaces (CLING/QUEL and CLING/SQL) [4],
(6) two versions of the toolkit,
(7) two versions of the framework, and
(8) an X11 and CLX[22] based version of the system.

The first version of the toolkit used a heavyweight abstraction for a field that was implemented by one or more X windows. For example, a labeled, type-in field used two X windows: one for the label and one for the field. A third window was created that contained these two windows when the form was being edited so the user could select the field with a mouse and move it to a new location. Unfortunately, the implementation of this abstraction was too inefficient.

The current toolkit provides widget, gadget, and synthetic-gadget abstractions that can be used to implement fields. Gadgets are used to implement display-only entities significantly reducing the number of X windows that have to be created and mapped when calling a PO. Decorative trim is an example of a display-only entity (e.g., the floor plan in the department panel). Synthetic-gadgets are display-list representations of drawable data that are used for high-performance output operations (such as tables and pop-up menus). They are similar to glyphs in InterViews [3]. Widgets are objects that are associated with X windows and can therefore directly process input events as well as display output. This version of the toolkit also implemented the geometry management abstraction and constraint system.

The first version of the application framework treated fields and variables as separate entities and implemented a customized propagation system to synchronize them. And, it used an interpreter to execute procedural code. The system was too slow. This problem was accentuated by performance problems that we encountered with early implementations of PCL which implements CLOS and the inherent speed of the Sun-3/75's and DEC Microvax-II's that we were using at the time.

The current framework treats field names as variables, directly executes procedural code, and uses a better implementation of PICASSO objects. The parameter passing model was also introduced in this version. Of course, performance was helped by the improved implementations of CLOS and the move to faster machines (e.g., Sparcstations). Run-time space is currently a problem since Common Lisp has never been particularly space efficient. The current implementation requires 16 to 32 megabytes of main memory for most applications. Our goal has always been to focus on functionality, speed, and rapid developmentrather than space. We believe the trade-off of space for CPU speed is reasonable since memory prices are rapidly decreasing. Furthermore, current Lisp systems are now providing tools to manage space more efficiently (e.g., the code Presto sharing and dynamic loading facilities of Allegro Common Lisp).

PICASSO has been used to implement several applications including:

(1) a facility management tool (CIMTOOL) [20][24],
(2) a recipe generator for process engineers

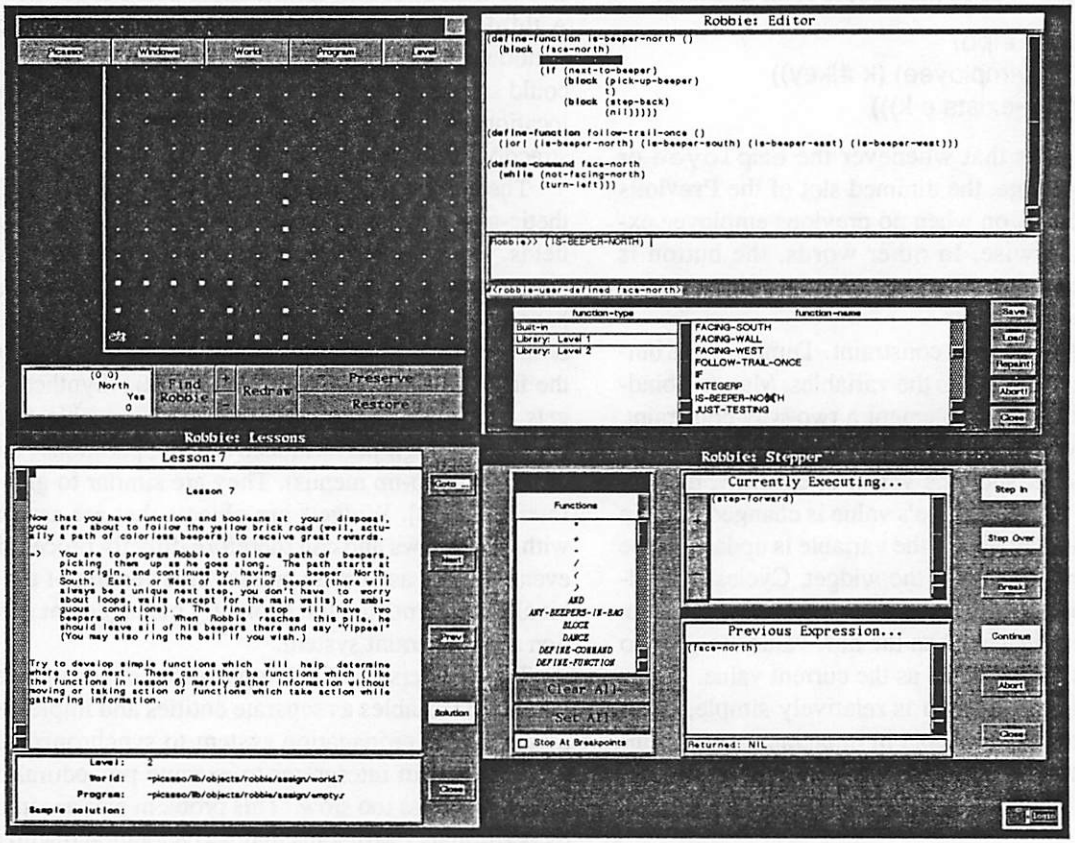


Figure 12: Robbie-the-Robot

(RGTOOL) [12],

(3) an educational program to teach Lisp programming (Robbie-the-Robot)[9],
(4) a direct manipulation interface builder,
(5) a hypermedia document editor and browser (HIP) [2], and
(6) an application for previewing and indexing video data.

A screen dump of Robbie-the-Robot is shown in figure 12. The frame in the upper-left shows Robbie in his two-dimensional world filled with beepers and walls. The panel in the lower-left displays the lesson on which the student is working. The panel on the upper-right displays a Lisp code browser and editor that highlights code as it is being executed. The panel on the lower right is a stepper-tracer that allows the student to control the execution of the program, to set breakpoints, and to browse the execution stack. This application was written in 2500 lines of code of which over 1000 lines were support code unrelated to the interface (i.e., the Robbie language interpreter), 750 lines defined new widgets (i.e., the graphical display and two widgets related to the editor/code browser), and 700 lines defined the interface using PICASSO. Robbie was written by two students over the course of a three week period (part-time) as a tool to perform experiments in programming education.

The PICASSO toolkit and framework contains approximately 35,000 lines of Lisp code of which only 3,000 lines are required to implement the framework. The first version of this system was distributed to adventurous users at other sites in early 1990.

A second release of the system is planned for later this year. It will include space optimizations and an improved geometry management and event handling system.

9. Conclusion

The PICASSO application framework provides higher level abstractions for creating user interfaces. Familiar programming language constructs (e.g., lexically scoped variables, call/return semantics, and parameter passing) are used to specify procedural code and control-flow. A simple constraint system is used to synchronize variables and the toolkit widgets through which they are displayed to the user and to trigger other code when appropriate.

Several applications have been developed using the toolkit and our experience thus far has been positive.

Acknowledgments

Many people have worked on the design and implementation of PICASSO. Dave Martin developed the XCL package and the original CLOS abstractions for the X Window System. Donald Chinn, Ken Whaley, and Scott Hauck worked on the early infrastructure and the first version of the toolkit. Scott Luebking extended the toolkit and implemented the first version of the framework. The current version of the toolkit and framework are major revisions of these earlier systems. We also want to thank our early users, including Beverly Becker, Jeff Goh, William Hunter, K.K. Lin, Lay-Peng Ong, Steve Smoot, and Kurt Partridge who have suffered from a buggy, slow system that never seemed to work as well for them as it did for us.

References

[1] P. S. Barth, "An Object-Oriented Approach to Graphical Interfaces", *ACM Trans. on Graphics 5*, 2 (Apr. 1986).

[2] B. S. Becker and L. A. Rowe, "HIP: A Hypermedia Extension of the PICASSO Application Framework", to appear in *Proc. NIST Advanced Information Interfaces: Making Data Accessible 1991*.

[3] P. R. Calder and M. A. Linton, "Glyphs: Flyweight Objects for User Interfaces", *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, Oct. 1990.

[4] D. Charness and L. Rowe, *CLING/SQL - Common LISP to INGRES/SQL Interface*, Computer Science Division - EECS, U.C. Berkeley, Dec. 1989.

[5] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, May 1983.

[6] D. Heller, *XView Programming Manual: An OPEN LOOK Toolkit for X11*, volume 7 in *X Window System Series*, O'Reilly & Associates, Sebastopol, CA, 1990.

[7] S. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.

[8] J. Konstan and L. A. Rowe, "Developing a GUIDE Using Object-Oriented Programming", to appear in *Proc. OOPSLA '91*, Phoenix, AZ, Oct. 1991.

[9] J. A. Konstan and B. Smith, *Robbie the Robot: Learning to Program in Lisp*, unpublished manuscript, Dec. 1989.

[10] J. A. Konstan, et. al., PICASSO *Reference Manual*, Computer Science Division - EECS, U.C. Berkeley, May 1990.

[11] G. E. Krasner and S. T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 Systems*, ParcPlace Systems, Aug. 1988.

[12] K. K. Lin, personal communication, Nov. 1989.

[13] M. A. Linton, "Composing User Interfaces with InterViews", *IEEE Computer*, Feb. 1989.

[14] J. H. Maloney, et. al., "Constraint Technology for User-Interface Construction in Thinglab II", *Proc. OOPSLA '89*, New Orleans, LA , Oct. 1989.

[15] D. C. Martin, *XCL - Common LISP X Interface (Protocol Version 10)*, Computer Science Division - EECS, U.C. Berkeley, Apr. 1987.

[16] D. C. Martin, *X/Common LISP Object System Interface*, Computer Science Division - EECS, U.C. Berkeley, June 1988.

[17] B. Myers, et. al., *The Garnet Toolkit Reference Manuals: Support for Highly Interactive, Graphical User Interfaces in Lisp*, Technical Report CMU-CS-89-196, Pittsburgh, PA, Nov. 1989.

[18] Next Corporation.

[19] Open Software Foundation, *OSF/Motif Programmer's Guide*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[20] L. A. Rowe and B. Smith, "A Facility Management Tool (Video Tape)", *DARPA/SRC CIM-IC Workshop*, Ann Arbor, MI, Aug. 1989.

[21] R. W. Scheifler and J. Gettys, "The X Window System", *ACM Trans. on Graphics 5*, 2 (Apr. 1986).

[22] R. W. Scheifler and O. LaMott, *CLX Programmer's Reference*, Texas Instruments, 1989.

[23] K. J. Schmucker, "MacApp: An Application Framework", *Byte*, Aug. 1986.

[24] B. Smith and L. A. Rowe, *An Application-Specific Ad Hoc Query Interface*, ERL Technical Report M90/106, University of California, Berkeley, Nov. 1990. *submitted to UIST '91*

[25] Y. Wang, *The Picasso Shared Object Hierarchy*, MS Report, Computer Science Division - EECS, U.C. Berkeley, June 1988.

[26] S. Wensel, *POSTGRES Reference Manual*, ERL Technical Report M88/20 (Revised), University of California, Berkeley, Apr. 1989.