# DEVELOPING A GUIDE USING OBJECT-ORIENTED PROGRAMMING

by

Joseph A. Konstan and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/93

11 October 1990

# DEVELOPING A GUIDE USING OBJECT-ORIENTED PROGRAMMING

by

Joseph A. Konstan and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/93

11 October 1990

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# DEVELOPING A GUIDE USING OBJECT-ORIENTED PROGRAMMING

by

Joseph A. Konstan and Lawrence A. Rowe

Memorandum No. UCB/ERL M90/93

11 October 1990

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# Developing a GUIDE Using Object-Oriented Programming†

*Joseph A. Konstan*
*Lawrence A. Rowe*

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

## Abstract

PICASSO is a graphical user interface development environment built using the Common Lisp Object System (CLOS). This paper describes how CLOS features including multiple inheritance, instance methods, multi-methods, and method combinations were used to implement the system. In addition, the benefits and drawbacks of CLOS development are discussed including code quality, maintainability and performance.

## 1. Introduction

This paper describes the object-oriented programming techniques used to develop the PICASSO graphical user interface development environment (GUIDE). The PICASSO system is composed of approximately 40,000 lines of Common Lisp [7] using the Common Lisp Object System (CLOS) [3]. Several programming techniques from the PICASSO implementation that take advantage of features in CLOS are presented. In addition, the benefits and drawbacks of using CLOS and these techniques for developing a GUIDE are discussed.

PICASSO is composed of an interface toolkit, an application framework, and a set of development tools. The toolkit provides the resources to create graphical user interfaces for the X window system [6] using Common Lisp. In addition to providing CLOS objects for standard X resources (e.g., windows, fonts, colors, graphics contexts, etc.), the system defines CLOS objects for toolkit widgets. Different types of widgets (e.g., text, tables, pictures and drawings) are implemented as classes in CLOS. A large library of predefined widget types is provided including: radio buttons, pop-up and pull-down menus, check boxes, scrolling tables and text, and graphics [4]. In addition, new widgets can be defined by PICASSO users and they can then be used interchangeably with the predefined ones.

---

† This research was supported by the National Science Foundation (Grants DCR-8507256 and MIP-8715557), 3M Corporation, and Siemens Corporation. Joseph Konstan was also supported by a National Defense Science and Engineering Graduate Fellowship granted through DARPA.

The **PICASSO** application framework [5] provides higher-level programming constructs for building applications. The framework provides abstractions for *forms* (the electronic counterpart of paper forms), *frames* (forms combined with pull-down menus to implement a major mode of an application), *dialog boxes* (modal interactors that are controlled with buttons) and *panels* (non-modal dialog boxes used to implement alternative views of the data displayed through a frame). CLOS objects are defined for these abstractions. These framework objects are called **PICASSO** Objects (PO).

The application framework also defines a data model for programming interfaces. PO's are like procedures in a conventional programming language. They may define local variables and constants, they are lexically scoped, and they may be called and

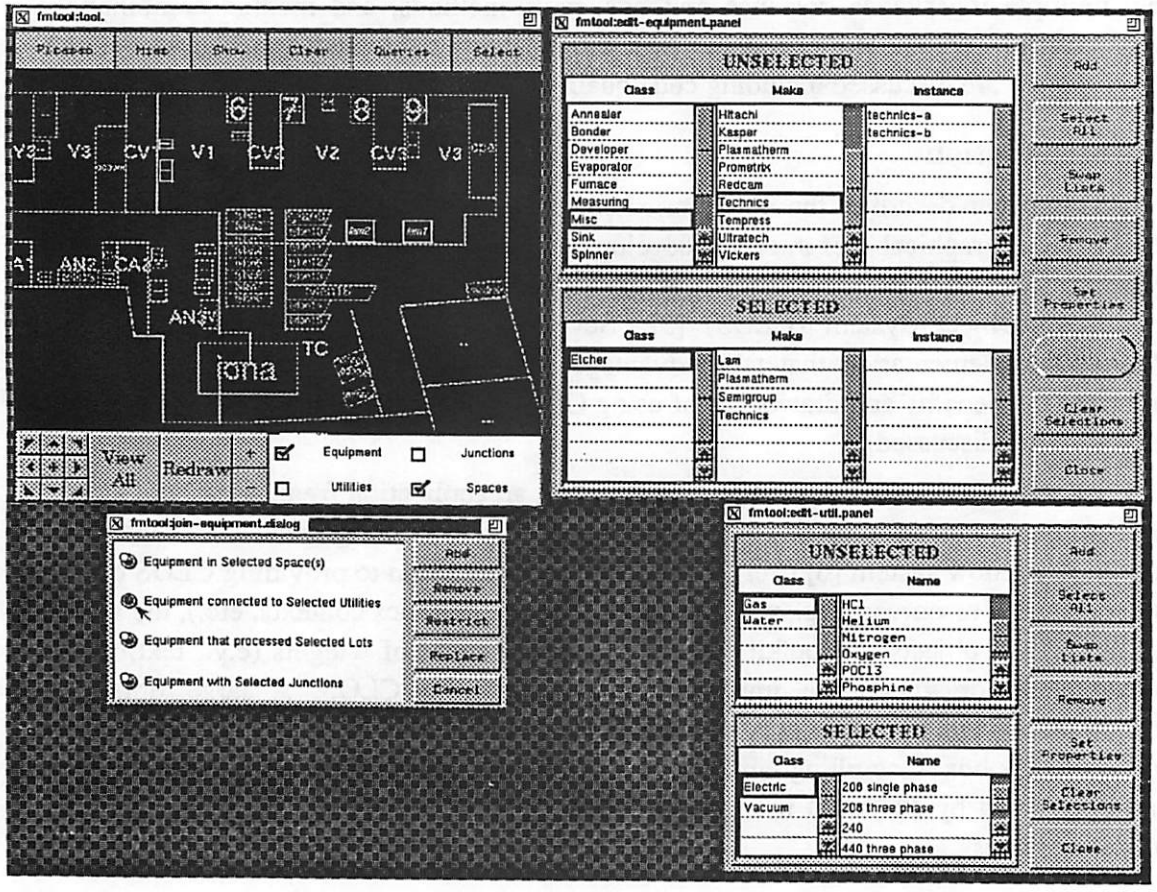


Figure 1: FMTOOL: A Tool for Managing IC-CIM Facilities

passed parameters. Five types of parameter passing are supported to implement different types of sharing including value, reference and value-result. Due to the asynchronous nature of user interfaces, PICASSO also supports value/update parameters (the callee is updated if the caller's variable changes) and value/update-result parameters.

A constraint system is provided that allows the programmer to specify arbitrary constraints between variables and object slots, and the propagation system keeps them up-to-date. For example, constraints are defined between variables in a PO and widgets that display their value to the user. Lazy evaluation of formula-constrained data is also provided to enhance performance. For example, a button that displays a certain rapidly-changing value when pressed can use lazy evaluation to avoid processing values that are not displayed.

PICASSO has been used to develop applications in semiconductor manufacturing and education. In addition to developing other applications, development continues on the system itself (e.g., a direct manipulation application builder) and on integrating hypermedia capabilities into applications (e.g., animation, audio, and full-motion video). A PICASSO application that displays and controls information about an integrated circuit manufacturing facility is shown in figure 1.

CLOS is an object-oriented programming system built on top of Common Lisp. It provides classes and methods found in most object systems along with a number of features not provided in simpler object systems. For example, CLOS supports: 1) multiple inheritance of both attribute slots and methods, 2) instance methods (i.e., methods specialized for a specific object), 3) multimethods (i.e., methods that discriminate on more than one argument), and 4) method combinations (i.e. factoring code into methods combined at run-time). This paper describes how these features were used to implement the PICASSO system.

The remainder of the paper is organized as follows. The next three sections discuss the use of multiple inheritance, instance methods, and method combinations. Section 5 discusses the benefits derived from and difficulties encountered using CLOS. And, section 6 presents conclusions.

## 2. Multiple Inheritance

This section describes several ways that multiple inheritance was used to simplify the PICASSO code. In CLOS, classes may inherit slot definitions, including slot attributes such as default values, and methods from any number of superclasses. If more than one parent defines a slot or method with the same name, an inheritance order, called a *class precedence list*, is used to determine the slot or method inherited. For a given class, the class precedence list is determined by the order in which the superclasses are specified in the class definition. This list is followed in a depth-first fashion, but search is cut off by a common superclass. For example, figure 2 shows a set of class definitions employing multiple inheritance. Figure 3 shows the class inheritance graph for these definitions. An instance of class **F** has three slots. `Slot-3` has a default value of 31, as

```
(defclass A ()
  ((slot-1 :initform 10)))
(defclass B (A)
  ((slot-1 :initform 11)))
(defclass C (B)
  ((slot-2 :initform 20)))
(defclass D (A)
  ((slot-2 :initform 21)))
(defclass E (C D)
  ((slot-3 :initform 31)))
(defclass F (D C)
  ((slot-3 :initform 31)))
```

Figure 2: Class Definitions

is specified in the definition of class **F**. `Slot-2` has a default value of 21, not 20, because **D** precedes **C** in class **F**'s class definition. `Slot-1` has a default value of 11, not 10, because **F** inherits from **B** before inheriting from the common superclass **A**.

Thus, in this example, the complete class precedence list for F is (F, D, C, B, A). In addition to using this order to search for methods and slot definitions, CLOS uses this list for `call-next-method`, the mechanism for invoking the same generic function in a superclass.

Multiple inheritance is used to improve code sharing among classes and to localize code that might need to be modified. Multiple inheritance was used in **PICASSO** to implement factored behaviors and abstract classes.

We use the term *factored behaviors* to refer to separating the different roles that objects play into different superclasses. For example, output and input behaviors are separated into the classes **gadget** and **widget**, respectively. A text gadget can display text but cannot receive input. A blank button that can be pressed, but displays nothing is a widget. Interface objects, which we generically refer to as widgets, inherit from both classes. Most text fields can be edited and therefore mix the behaviors of the display-only text gadget class with the editing behaviors of the text widget class. And, most buttons mix display behavior (e.g., displaying a text label or picture in the button) with input behavior (e.g., detecting and responding to mouse button presses). Titles and other decorative trim in a form cannot be changed by the user so they can be implemented by a gadget.

The first implementation of **PICASSO** did not factor input and output behaviors into different classes. As a result, it was difficult to improve the performance of widgets that did not need all of the input behaviors associated with the class widget. Specifically, the

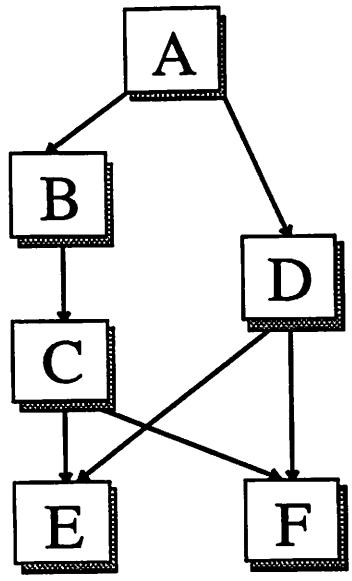


Figure 3: Class Inheritance Graph

performance of text labels and of menus was unacceptably slow. Factoring the behaviors allowed us to separate out the more costly event-handling and input-oriented behaviors and to use lightweight text and picture gadgets for higher performance.[1]

It is virtually impossible to implement cleanly factored behaviors without multiple inheritance. A single inheritance system forces the programmer to specify descendant behavior as a customization of a single parent. PICASSO widgets, however, tend to inherit from at least two superclasses, both of which define many slots and methods. Figure 4 shows the class hierarchy for some typical widgets. A text-widget, for example, inherits behaviors from the **text-gadget** class that displays text and from the **widget** class that incorporates all of the behaviors of X windows (e.g., event handling). In single-

[1] Later, we introduced synthetic gadgets for very-high performance areas where even the weight of a gadget was too high. These synthetic gadgets are little more than display lists with the correct methods defined. This iterative process of continual factoring for the sake of performance has happened several times in the development of PICASSO.

inheritance systems, one of these behaviors would have to be incorporated into text-widget in a different way.

Factoring behaviors produces classes that can stand alone (e.g., they can be instantiated) and combined together to mix their behaviors. However, sometimes classes are created that will never be instantiated. They define code used in other classes. We call these classes *abstract classes*. Two reasons for using abstract classes are to modularize code and create mixins.

Abstract classes helped modularize the code behind the PICASSO application framework. Figure 5 shows the class hierarchy for the PO classes. The classes **picasso-object**, **top-level-po**, and **callable-po** are never instantiated. Instead, **picasso-object** holds code common to all PO's (e.g., call and return semantics, lexical and variable-holding behaviors, and grouping behaviors inherited from **collection-widget**). **Top-level-po** adds the special behaviors needed by PO's that are displayed directly on the root-window as opposed to being contained inside other PO's (i.e., tools, panels, and dialog boxes). **Callable-po** adds the behaviors of PO's that are called like functions and co-routines (i.e., frames, panels, and dialog boxes). By using abstract classes and multiple inheritance, these behaviors are separated into distinct modules. Otherwise, code would either

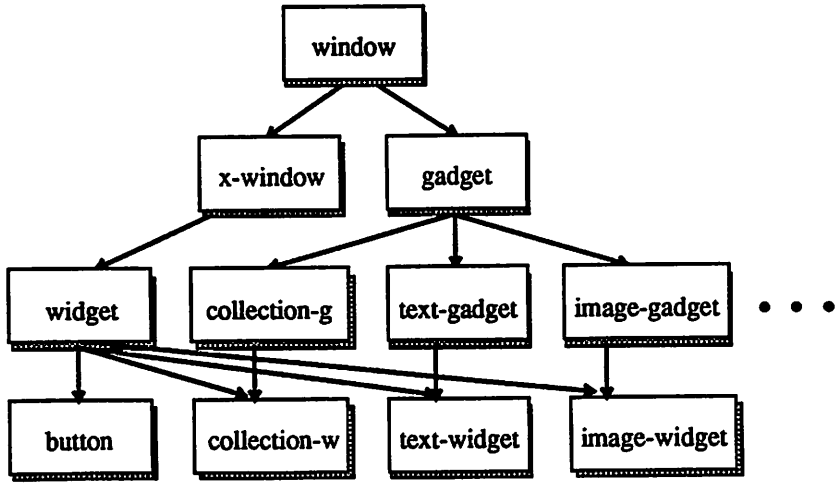


Figure 4: Widget/Gadget Class Hierarchy

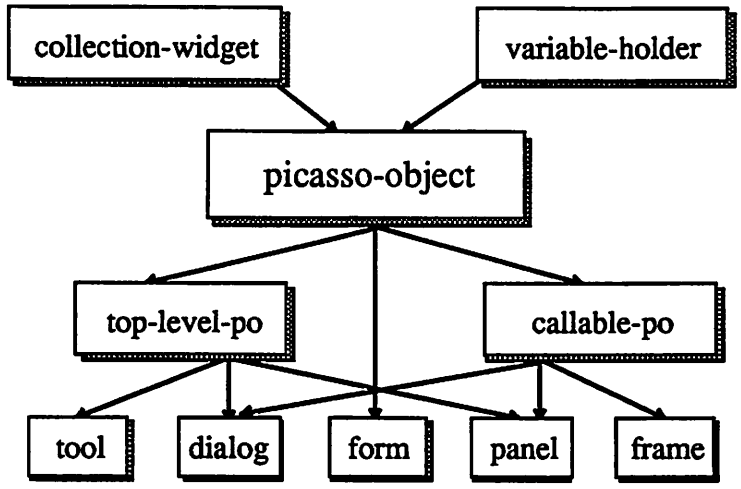


Figure 5: PICASSO Object Class Hierarchy

have to be duplicated (by placing the common behaviors in each **picasso-object** subclass) or moved into the superclass **picasso-object** and then selectively enabled when needed. Duplicating code causes maintenance problems and moving code into **picasso-object** hurts both performance and code maintainability by making superclass behaviors dependent on their subclasses.

Mixins are behaviors that can be added into any of a number of other classes. They are "mixed in" by creating a new class that inherits from both the original or base class and the mixin class. Mixins have two major benefits in developing a GUIDE such as PICASSO. First, they allow behaviors to be shared among classes that are otherwise distantly spread across the class hierarchy. Second, they allow easy prototyping of situations where behaviors may or may not belong in certain classes. As an example, the behavior of holding variables (and therefore being a lexical entity) in PICASSO is defined by the mixin class **variable-holder**. It was clear through most of the development of the framework that PO's should be variable holders, but it was not clear whether any other entities should also be variable holders. Specifically, collections which hold other widgets for grouping behaviors and tables could arguably benefit from holding variables. For development purposes, variable holding behaviors could easily be mixed into these classes to further research this question.

Designing mixin classes is trickier than designing ordinary classes because an effective mixin should not disturb the other operations of the class it is being mixed into. A mixin must not conflict with the slots and methods that may be defined in any class except where necessary for the operation of the mixin. In this case, **variable-holder** required a single slot to hold the variable table and a single accessor method for that table. The rest of the class hierarchy was not permitted to use that slot or method name.

Multiple inheritance greatly simplifies the development and maintenance of a large system such as PICASSO. There are some cases, however, where multiple inheritance is too cumbersome to use. The main disadvantage of mixins is the combinatorial explosion in the number of classes that must be defined if all of the behaviors defined in the mixin classes can be combined orthogonally. This large number of classes reduces the maintainability of the code by requiring developers to understand a great number of classes and to code all mixins orthogonally to prevent conflicts in common descendants. A large number of classes is also inefficient since the creation of a class than may not be needed wastes both processing time and memory. Some object systems support dynamic classes which are instantiated at run-time as needed. Dynamic classes remove the run-time inefficiency of a large class space by allowing classes to be specified as a list of superclasses. The next section introduces an alternative solution to this problem that uses instance methods.

## 3. Instance Methods

Instance methods discriminate on the value of an argument(s) rather than the class of the argument(s). They define a behavior for a single instance of a class. Slot-value methods define a behavior for all instances of a class that have a specific value in a particular slot. CLOS provides an *eql* method structure that can be used to implement instance and slot-value methods.

Instance methods are used in PICASSO to implement the propagation system that constrains slot values to be equal to the result of specified functions of other slot values.[2] The Lisp form **setf** is used to set slot values with the expression:[3]

`(setf (`*slot-name object*`)` *new-value*`)`

This form invokes a method named `(setf` *slot-name*`)` that takes the new value and the object as arguments and discriminates on the class of the object. `Setf` methods can be written just as any other methods.

---

[2] Propagation also constrains PICASSO variables, but since variables are implemented as CLOS objects the same mechanism can be used.

[3] Technically, `setf` is a macro that expands into functions and methods depending upon the target being set.

A simple implementation of propagation can use the `setf` method for all classes to check whether the change requires a propagation to occur when any slot value is set. This solution is inefficient, even with caching, since setting a slot value must be a fast operation and relatively few slots have propagations that depend on them. By using instance methods, the propagation behavior is added only to the specific slots of objects that need to propagate their changes. These are the slots and objects referred to as arguments in the function used to constrain another slot.

In this case, the pre-existing setf method for an object slot is augmented by defining an **eql** *around* method. Around-methods, discussed in more detail in section 4, wrap themselves around primary methods. They are invoked first, and the primary method is called under their control. After the primary method returns, control returns to the around-method. In other words, the around-method specifies code to execute before and after the primary method. For propagation, the following method is dynamically defined for any object slot that must be propagated:

```
(defmethod (setf slot-name) :around (value (self (eql object)))
      (unless (equal (slot-value self 'slot-name) value)
          (call-next-method)
          (propagate (gethash unique-key *prop-table*))))
```

The argument list for this method indicates that it applies for any value, but only for the specific object designated. The body of the method checks first to assure that the slot value indeed changed to cut off loops. Then, the primary method is called to update the slot value. Notice that this approach insures that any error checking, side effects, or other processing will be done. Once the primary method returns, the around-method calls the propagate function to pass on the changes to whatever slots have registered interest in this slot value.

Custom `setf` methods are used for virtual slots and PICASSO variables. Virtual slots are implemented as methods to access and update a value without actually storing the value. The `setf` method does not check for a change since no old value can exist. PICASSO variables, for efficiency, always propagate since most variables have propagations. This example illustrates another benefit of instance methods: different variants of the method can be defined to optimize cases that do not deserve their own classes.

Slot-value methods are also used to overcome the problems of combinatorial class explosion introduced by multiple inheritance. The different slot values define a set of *virtual classes* each of which has the same slots and class-discriminating methods but different slot-value methods. The following two examples show how virtual classes reduce the number of classes in the system and make changing classes faster and easier.

Geometry management is the process of sizing and laying out windows within a parent window. This process is implemented in PICASSO by a geometry manager that is bound to a collection. A geometry manager includes a data structure that holds layout hints and a collection of routines that pack children within a collection when called. A geometry manager also has routines that respond to asynchronous changes to the children

in the collection (e.g., adding or removing a child), requests from children for different sizes, and notifications that the collection itself is being resized. Approximately ten geometry managers are provided in PICASSO (see figure 6) and new ones can be added by defining the appropriate functions.

The obvious implementation defines an abstract class for each geometry manager and mixes that class into the base collection classes to yield a different class for packed-collection-gadget, rubber-sheet-collection-gadget, etc. This approach leads to 30 or 40 new classes and even more classes if subclasses of collections (e.g., form, table, etc.) can have any of these geometry managers. It also makes it more difficult for a user-defined geometry manager to be fully incorporated into the system because the user must add many new classes.

CLOS provides two solutions to this problem. One solution is to create these classes dynamically as they are needed. Classes can be dynamically created rather easily with a meta-class protocol feature that allows a class to be inserted into another class' superclass list. While this solution is a perfectly reasonable implementation, different solution was used in PICASSO because this solution results in less obvious code and greater difficulties when changing a collection's geometry manager.

| Name | Function |
|---|---|
| anchor-gm | controlled stretch and relative positioning |
| just-pack-gm | full-width menu bars |
| left-pack-gm | compressed (pushed left) menu bars |
| linear-gm | linear stretch (useful for bordered objects) |
| matrix-gm | tabular layout |
| menu-gm | layout of menu entries |
| null-gm | default, just places objects where they request |
| packed-gm | perpendicular packing (useful for fill-in forms) |
| root-gm | special manager for PICASSO root window |
| rubber-gm | rubber sheet (complete stretch) |
| stacked-gm | vertical and horizontal stacking for icon palettes |

Figure 6: Geometry Mangers Defined for PICASSO

The PICASSO solution uses slot-value methods to call the appropriate geometry manager. Since each collection has exactly one geometry manager, we include a slot in the collection that holds the name of the geometry manager (i.e., a Lisp symbol). The methods that implement a specific geometry manager discriminate on the value stored in this slot, rather than on the class of the object passed to them. For example, the method that handles repacking a collection is defined as

```
(defmethod gm-repack ((gm (eql 'my-gm)) self)
           code)
```

This method is called when the first argument is the symbol `my-gm`. To make it easier to program this way, we add a simple macro to handle passing in the slot-value:

```
(defmacro repack (self)
   `(gm-repack (gm ,self) ,self))
```

Thus, we can call `repack` as if it were an ordinary method, passing only the collection as an argument, and it will call the correct repack method.

The other difficulty is to allocate storage for the geometry manager to use, since virtual classes are not real classes so they cannot add slots. For geometry managers, the solution is straightforward. All geometry mangers are defined to use certain slots that are present in all collections: 1) a **children** slot that holds a list of child windows for which the geometry manager is responsible (in an order it manages) and 2) a **gm-data** slot that holds other data including layout hints and cached results. The geometry manager routines are given complete control of this **gm-data** structure, and they can use it for any purpose.

A second use of slot-value methods in PICASSO is for widget borders and labels. A border describes the graphics that surround a widget to enhance its visual appearance. Many borders are provided including drop-shadows, picture frames, and boxes. A label contains text or an image that identifies the widget. They can be positioned in various locations including to the left, above, or below the widget or in a smaller font in the frame. Figure 7 shows some of the borders and labels provided with PICASSO. Users can define additional borders and labels by naming them and defining appropriate methods.

Labels and borders are implemented using the same technique we used for geometry managers. Since the data structures used by borders and labels are better defined, they are given more detailed slot assignments including **label-x**, **label-y**, **label-position**, **label-value**, **border-type**, and **border-width**. When no border or label is desired, there is a small space penalty for these extra slots but no method is executed so the time penalty is insignificant.

There are several benefits to using slot-value methods as an implementation of virtual classes. First, they are easy to implement and extend. For example, a new geometry manager, label, or border can be added by selecting a name and defining a couple of methods. The performance penalty when used is small (i.e., the cost of accessing the

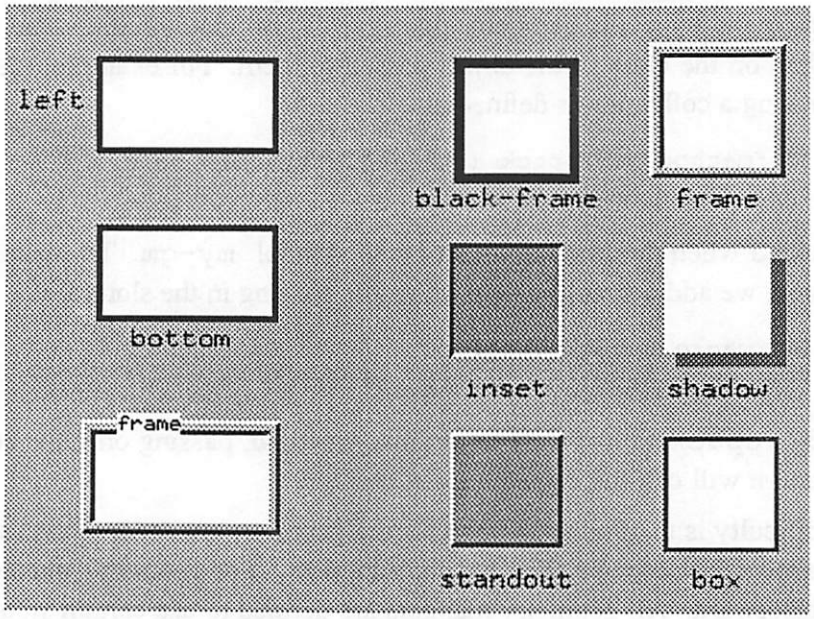


Figure 7: PICASSO Borders and Labels

slot-value before method discrimination) and the performance penalty when not used is small (i.e., the unused extra space). And, it is easy to change the virtual class of an object by changing the value in the appropriate slot.

With these benefits come some limitations as well. The two biggest limitations are the inability to define slots in virtual classes and the very limited inheritance available. Since virtual classes are not classes, they can only use the slots defined in the classes into which they are mixed. This limitation requires that all classes for a given virtual class must use the same slots which typically limits this technique to small features implementing different versions of the same attribute. Additionally, the inheritance available for virtual classes is minimal. Since they are based on symbol equality, there is a very strict two-level virtual class tree. At the root is class **t** that applies to everything and at the leaves are all of the virtual classes.[4] As a result, virtual classes implemented this way do not work well when there are large amounts of code to be shared among some, but not

all, of the variants.

## 4. Multimethods and Method Combinations

As we have seen, CLOS allows very powerful method constructs that can discriminate on both class membership and equality. This section describes how PICASSO used CLOS multimethods and method combinations.

PICASSO has tended to avoid using multimethods due to performance considerations which are discussed in the following section.[5] Multimethods have been used, however, to prototype behaviors that were later implemented in other ways or in some cases abandoned. Two examples are the development of type-sensitive widgets and methods that handle different types of windows.

A feature tested in an early version of PICASSO was a type-sensitive widget. This type of widget would display only certain types of values (e.g., integers, strings, or arrays). The widget would change itself into a different widget if it was set to a value that it was unable to display. In this way, a single widget could be created to display a numerical value. If, for some reason, a picture was assigned to that widget, it would automatically change itself into a **picture-widget**.

Implementing this type of automatic class changing was simplified by writing `setf` methods that discriminated on the type of the value as well as the type of the widget. There would be a method to set meters to integers and floating point numbers but not to strings. The default method for widgets would then change the widget into an appropriate type of widget. The performance of multimethods was not a problem in this situation because this operation was executed infrequently and changing classes was already slow. However, we abandoned this idea for a more flexible synthetic gadget.[6]

---

[4] Another possible implementation would create a class for each geometry manager, label or border and place an instance of that class in a slot in the collection or widget. This implementation solves the slot and inheritance problems but is no longer a lightweight implementation. Indeed, it merely adds a list of components to an object, each of which has its own methods, with methods on the holder that invoke methods on the proper component. We initially chose not to employ this implementation but are now considering implementing geometry managers, labels, and borders in this fashion.

[5] In an earlier system we tried to use multimethods to implement event dispatching. A generic dispatch function was defined that took an event and window object and discriminated on both. Besides learning that multimethods were not implemented correctly in the early CLOS implementation we were using, we also quickly realized that method determination for multimethods was too slow for event dispatching.

[6] A synthetic gadget contains only data and a list of display parameters. The method `put` is defined on each data type to paint the data onto the screen. Where a text gadget is an object with many slots representing all of the possible functionality for text and windows, a text synthetic gadget contains only a

PICASSO still uses multimethods for a few cases where operations depend on two different widgets or gadgets. In some cases, there is a simple X server call that can perform operations on two X windows (e.g., calculating relative coordinates or positions in the window hierarchy) but does not operate on non-X-windows (e.g., gadgets and synthetic gadgets). In these cases, a method is defined that discriminates on the class of both objects. If they are both X windows, the server call is performed. Otherwise, toolkit code is executed to perform the operation.

Multimethods could be replaced in all cases with code that resembles a case structure. Inclusion of multimethods allows programmers to take advantage of the built-in CLOS method dispatching, with its caching and other performance tuning, rather than writing ad hoc, and likely less performance tuned, custom dispatchers.

In CLOS each generic function has a primary method as well as before-, after-, and around-methods. These additional methods layer their execution on top of the primary method. Suppose that we have two classes **super** and **sub** and that a method `foo` has a primary, before-, after-, and around-method on each class. Figure 8 shows the definitions for class **sub**. The definitions are the same for class **super** except that the formatted print statements read "SUPER" and the primary method does not execute the `call-next-method` call.

```
(defmethod foo ((self sub))
  (format t "Entering SUB Primary Method")
  (call-next-method)
  (format t "Exiting SUB Primary Method"))

(defmethod foo :before ((self sub))
  (format t "SUB Before Method"))

(defmethod foo :after ((self sub))
  (format t "SUB After Method"))

(defmethod foo :around ((self sub))
  (format t "Entering SUB Around Method")
  (call-next-method)
  (format t "Exiting SUB Around Method"))
```

Figure 8: Method Definitions for Class **sub**

string, a location for painting, a font, and some colors.

When the method `foo` is called on an instance of **sub** the output shown in figure 9 is produced by the `format` functions. In each case where `(call-next-method)` appears, not executing that expression would result in skipping forward to the corresponding "Exiting" clause without executing any additional methods in between. For example, if the around `foo` method for **sub** did not execute `call-next-method`, none of the other methods would be called.

Before-methods execute before any primary methods. Before a primary method is executed, *all* before-methods that apply are executed *from most to least specific*. Even if a superclass' primary method is not executed, its before-methods are always executed. Therefore, before-methods should only be used when any possible subclass will also need the same behavior.

In PICASSO, before-methods have a natural place in implementing lazy evaluation slots. These slots are typically defined for a class, although they can also be defined for an instance. Lazy evaluation slots check a cache stored in the slot for validity when the slot is accessed. If the cached value is valid, it is returned. If not, the cached value is recomputed. The cache is automatically invalidated when appropriate. It is assumed that there may be primary methods on the slot to properly convert data or perform side effects. This lazy slot behavior is implemented with the following before-method:

```
(defmethod slot-name :before ((self class-name))
  (when (invalid-p (slot-value self 'slot-name))
      (setf (slot-value self 'slot-name) recomputation formula)))
```

The body of the method uses the CLOS accessor `slot-value` to avoid recursively

```
Entering SUB Around Method
Entering SUPER Around Method
SUB Before Method
SUPER Before Method
Entering SUB Primary Method
Entering SUPER Primary Method
Exiting SUPER Primary Method
Exiting SUB Primary Method
SUPER After Method
SUB After Method
Exiting SUPER Around Method
Exiting SUB Around Method
```

Figure 9: Call Sequence for Method Combinations

calling this method or a `setf` method. This technique is common in before-methods that wish to prepare the data without getting trapped in an infinite loop. This implementation of lazy slots prevents the slot accessors themselves from having to know that the slot is lazy. Instead they can assume that whenever they are called, the correct value is there.

After-methods execute after all primary methods. If one or more primary methods have executed, *all* after-methods are executed *from the least to most specific*. This order is the opposite of the order in which before-methods are executed. Again, all after-methods are executed if any primary method is executed, so they should only be used when any subclass will need the same behavior.

After-methods are used in **PICASSO** to implement side effects that require a fully initialized object. As an example, the `new-instance` method, which is called to initialize a new instance of a class, for collections has an after-method that creates the children objects in the collection. It is more efficient to wait until the collection is properly initialized before creating the children objects, so an after-method is ideal. After-methods are also defined on `new-instance` to perform other side effects such as informing the geometry manager that a new widget has been added to a collection. These side effects are best handled after the object has been properly initialized.

Around-methods wrap behaviors around the rest of the methods. In structure, they are much more like primary methods than before- or after-methods. When a method is invoked, the most specific around-method is called even if there is a more specific primary, before-, or after-method. If an around-method calls `call-next-method` the next most specific around-method is called. If and when the most general around-method calls `call-next-method`, all of the before-methods execute, followed by the most specific primary method and any more general primary methods if it calls them and then all of the after-methods are executed. At this point, control returns to the most general around-method and back up the around-methods as each returns.

Around-methods are used in **PICASSO** to prevent primary methods from executing. Section 3 discussed an example in which an instance around-method prevents the primary `setf` method from executing if no change has occurred. Around methods are also useful because they can return values. In some cases, such as the creation and invocation of PO's, around-methods are used to allow values to be correctly returned when they cannot be computed until after all after-methods have executed.

Method combinations have another use when combined with bushy abstract class hierarchies. Proper use of method combinations allows the maximum sharing of code. Using only primary methods, a subclass and superclass have three phases of execution (subclass before call-next-method, superclass, subclass after call-next-method). Adding before-, after-, and around-methods provides the twelve phases shown in figure 9. The method `invoke` for PO's, which calls a PO, is defined in eleven pieces. It handles parameter passing, allocating lexical children, managing the display, and event handling.

| Class | Type | Description |
|---|---|---|
| picasso-object | before | handles in-use objects, allocates resources |
| picasso-object | primary | processes parameters, allocated local vars, resolves references to external objects |
| picasso-object | after | notify parent about self, execute setup and init code |
| top-level-po | before | places PO on root window |
| top-level-po | after | handles mouse warping and delayed exposure |
| callable-po | primary | processes contained form variables |
| callable-po | after | invokes contained form |
| tool | around | handles package search list, calls start frame, starts event loop |
| dialog | around | positions dialog over caller, starts event loop |
| form | after | exposes window |
| frame | around | handles nested calls, starts event loop |

Figure 10: Invoke Methods for PICASSO Objects

The table in figure 10 shows the different methods defined for `invoke`. Figure 11 shows the order in which these methods are called when a PICASSO frame is called. Recall that the class precedence list for **frame** is **frame, callable-PO, picasso-object**.

Using method combinations to create layered behaviors has benefits and drawbacks. The main benefit is that more code can be implemented once for the class **picasso-object** rather than several times. The biggest drawback is that the implementation is very complicated and requires a clear understanding of the intent of each phase of the method calls. The original implementation of PO's did not use these layered methods. As a result, much of the code that was shared by different PO classes (e.g., notification of parents, invoking contained forms, and resource allocation) could not be placed in the superclass methods due to execution order constraints. Consequently, the code was copied into the methods for each PO class which made maintenance difficult. This poor design was so difficult to develop further that we redesigned the PO class hierarchy. By virtue of our prior experience, we were able to see the actions that depended on other actions and developed a cleaner layering of behavior.

| Class | Type | Description |
|---|---|---|
| frame | around | check for and conceal existing current frame<br>set invoked frame's parent to the current tool |
| picasso-object | before | check to see if frame is in use<br>allocate X resources for frame |
| callable-po | primary | no action taken before call-next-method |
| picasso-object | primary | process all variables and parameters<br>allocate lexical children |
| callable-po | primary | establish frame-level aliases for form variables |
| picasso-object | after | notify lexical parent of frame's invocation<br>execute frame setup code and initialization code |
| callable-po | after | invoke frame's form with appropriate arguments<br>which recursively calls `invoke` on a **form** object |
| frame | around | put frame on call-stack and start event loop |

Figure 11: Invocation of a Frame

Multimethods and method combinations make it possible to write very compact, modular code that takes full advantage of the object system's built-in method dispatcher. Performance, of course, will depend on the implementation of the object system.

## 5. Discussion

This section discusses the impact CLOS had on the development of PICASSO. CLOS made PICASSO far easier to prototype and develop. On the other hand, it complicated the system and makes the system harder for new researchers to change. This section also discusses some performance issues encountered during development.

Without question, CLOS made prototyping and implementing new features in PICASSO fast and easy. The entire constraint system, including propagation and triggers, was implemented in 350 lines of code. The lazy evaluation slots referred to above were implemented in 50 lines of code. The entire application framework (including all PO's, the lexical environment, and PICASSO variables) was implemented in under 2000 lines of code. We estimate that writing the framework and toolkit without CLOS, just in Common Lisp, would require about twice as many lines of code. The CLOS features discussed in this paper (i.e., multiple inheritance, instance methods, and method

combinations) have saved 5000 to 10,000 lines of code and their use resulted in a cleaner implementation.

For the most part, CLOS has also been a great benefit when adding new features and prototyping changes. Method combinations have made it easy to prototype changes and experiment with new ideas. Multiple inheritance allowed us to implement widgets such as radio-button groups in under 100 lines of code.

With all this saved code and the benefit of the class abstraction, you might infer that CLOS made PICASSO's implementation easier to understand. In fact, the opposite was more often the case. Multiple inheritance required each superclass, and almost any class should expect to become a superclass, to be designed to share superclass responsibilities. For example, each method had to invoke `call-next-method` even if the superclass had no next method, since a subclass might inherit this method from two superclasses and `call-next-method` is the way for the second superclass' method to be invoked. Consequently, many methods had to be defined on the class at the top of the hierarchy (i.e., **window**) to serve as placeholders. These methods are required because subclass methods that `call-next-method` generate an error if no method is available. Of course, you could argue that these methods are actually prototype definitions for optional methods defined lower in the class hierarchy and that they should be defined in the superclass to document the abstractions. Teaching developers to design clean methods for multiple inheritance took some effort, but good programmers were able to write such methods with a couple of week's practice.

The next problem was that the CLOS model of inheritance does not support or encourage encapsulation. As a result, all behaviors of all superclasses have to be well-understood before writing a new subclass. We discovered that conventional documentation did not address this problem. An interactive, dynamic form of documentation that indicates non-overridden inherited behaviors in the documentation of each child is needed. Moreover, a good development environment should provide an interactive object inspector and class hierarchy browser similar to the tools provided by Genera [8] or SmallTalk [2]. hierarchy.

A final difficulty with multiple inheritance is that the class inheritance order matters. While this concept is not difficult to understand, many of our methods are order-dependent and we found that avoiding circular dependencies was often difficult. As a result, method combinations were used to isolate explicitly layered behaviors.

Instance methods presented almost no problems for our developers. While most programmers had not heard of them, they were easy to understand and use. Indeed, instance methods turned out to be the one feature of CLOS that simplified code and made it more compact.

Method combinations, even more than multiple inheritance, made the system harder to understand and modify. The layers of abstraction must be well-understood and conventional documentation was inadequate.

The final serious problem we had using CLOS is ironically problematic with research development. Since CLOS does little to support or encourage encapsulation of superclass features, each detail of the superclass implementation is quite visible to the subclasses. In an existing system, where superclasses towards the root are unlikely to change, this design works well. However, in developing PICASSO we found that major changes were being made to these base classes rather frequently. Most changes to a base class required rewriting code in subclasses that inherited from the class being changed particularly when the changes involved adding or removing slots and methods. This effect is partly a product of poor object-oriented design, partly unavoidable, but partly attributable to CLOS.

The performance of Common Lisp and CLOS continues to be a big concern because the success of a graphical user interface can be determined by the perceived responsiveness of the system. We have been using a portable implementation of CLOS developed at Xerox (PCL). [1] We recognize that some of our performance concerns will be addressed by native implementations of CLOS. Nevertheless, the success of PICASSO will be to some extent determined by the performance of Common Lisp and CLOS.

The two major performance concerns are space and time. There is no question that Common Lisp and CLOS cost us a great deal of space. On a Sun Sparcstation computer using Allegro Common Lisp, the size of the Lisp image is over 5 megabytes on disk. Adding CLOS, the CLX interface, and some database access code brings this to over 8 megabytes. The PICASSO toolkit and application framework add another 5 megabytes which brings the disk image of PICASSO to 13 megabytes.[7] Adding in application code increases the space. The CIM Facility Browser tool adds another 2 megabytes to this when packaged as an application. On a positive note, the run-time image of PICASSO rarely exceeds 16 megabytes which indicates that the system does not grow much when executing.

We recognized that a Lisp system would be larger than a similar system written in C when the project started. For example Windows/4GL, a commercial system written in C that uses the X Toolkit and OSF/Motif look and feel, duplicates some of the functionality of PICASSO in under 4 megabytes. We estimate that a complete implementation of PICASSO in C would result in an image size of 6 to 8 megabytes.[8] Clearly, space is a problem but we are willing to trade space for power, as long as speed was not an issue.

---

[7] The disk space used is highly dependent on the specific machine architecture and the quality of the compiler. For example, Sun 3 and Sequent Symmetry images of PICASSO use about 11 megabytes and Decstation 3100 images use 18 megabytes.

[8] We could shrink the present PICASSO image by about 2 megabytes by removing the compiler and other development tools. We include these tools because PICASSO is designed to be an extensible system and they are important for building extensions to the system.

Runtime performance is largely determined by the time it takes to do a method call. Ironically, the method call time is not a performance bottleneck because method combination lists are cached and a high percentage of methods called are in the cache. The cache reduces the time to call a method to approximately 2.5 times the time required to call a function.

The biggest performance problem we experienced was with keyword parameters to functions and methods. The Common Lisp keyword mechanism requires that keywords be reparsed for each function called. In particular, every `call-next-method` reparses the keyword parameters. This performance penalty is substantial because we use many keyword parameters so that applications can selectively override default values (e.g. creating a text widget calls approximately 70 methods which have an average of 30 allowable keyword parameters).

We have removed keyword parameters from many run-time critical methods and functions to improve performance, but we still pay a significant overhead on object creation. A solution to this problem would be an automatic system to normalize methods and method calls. This normalization would define a unique ordering of keyword parameters for any function or generic function. Then, the compilation of a method or method call would automatically rearrange the actual keyword arguments to match this unique ordering. Interpreted methods and method calls would still require keyword parsing but compiled methods and method calls would not. We have not developed such a normalizer but expect that a Lisp implementer will have to do so to stay performance-competitive.

A final performance consideration is the compilation of methods generated at run-time. Triggers, propagation, and some instances of lazy evaluation require that new methods be defined at run-time. Portable implementations of CLOS make it very difficult to compile these methods on the fly. Any native implementation will need a simple compilation function for methods to improve the performance of methods defined at run-time.

In summary, most of the programming techniques discussed in this paper do not significantly degrade performance. Multiple inheritance could cause problems with method resolution time, but caching of method combination lists minimizes the overhead. Instance methods are a tiny bit slower than class methods (approximately five percent) but this time is still better than the cost of dispatching the various behaviors. Method combinations have lowered system performance due to keyword processing costs and the basic overhead of method calls. In addition, the lack of code duplication has caused the size of the PICASSO image to shrink as refinements were made to layer behaviors which presumably improved performance.

## 6. Conclusions

Using CLOS to develop the PICASSO GUIDE resulted in faster development, easier prototyping, and more modular and compact code. Taking advantage of CLOS features created complex interactions among classes and methods that makes it hard for a new

developer to learn the PICASSO implementation and makes certain modifications difficult. The details discussed here are hidden from users who develop and use PICASSO applications. And, once a developer has learned the implementation, he or she reaps the benefits of CLOS and is able to accomplish a great deal.

This paper presents some programming techniques using CLOS that are applicable to other areas. First, instance methods are effective ways of specifying instance-specific behaviors and implementing slot-value methods for lightweight virtual classes. Second, method combinations are an effective way to reduce code duplication by layering behaviors in a cluster of classes. And, mixin classes, when properly designed, can make experimenting with new behaviors easy and they can make code much easier to read.

Lastly, several areas that need more work were identified. First and foremost is the development of a sophisticated environment for CLOS development. Object systems in general create documentation problems. For a programmer, a tool is needed to browse the class hierarchy and a full code walker is needed to recognize which inherited behaviors are included and which are pre-empted at a specific place in the code. And, work needs to be done on the performance of method calls to make CLOS competitive with object systems based on C. This problem is clearly a case where individual vendors must part with portability while adhering to the standard to achieve optimal performance.

## Acknowledgements

Many people have worked on the design and implementation of PICASSO. David 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. Brian Smith developed much of the present PICASSO toolkit design and implementation and he developed the CIM facility browser shown in figure 1. Steve Seitz implemented many of the performance enhancements for the toolkit as well as the label and border abstractions. Chung Liu developed many PICASSO widgets and early applications. We also want to thank our early users for their patience and feedback, especially Beverly Becker, who developed a hypermedia system and is now adding hypermedia features to the toolkit.

## References


1. D. Bobrow and G. Kiczales, "Common Lisp Object System Specification", Draft X3 Document 87-001, Am. Nat. Stand. Inst., February 1987.
2. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, May 1983.
3. S. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.
4. J. A. Konstan and et. al., "PICASSO Reference Manual", *Electronics Research Lab. Technical Report M90/79*, Sep. 1990.
5. L. A. Rowe and et. al., "The PICASSO Application Framework", *Electronics Research Lab. Technical Report M90/18*, Mar. 1990.
6. R. W. Scheifler and J. Gettys, "The X Window System", *ACM Trans. on Graphics 5*, 2 (Apr. 1986).
7. G. L. Steele, *Common Lisp - The Language*, Digital Press, 1984.
8. J. Walker, D. Moon, D. Weinreb and M. McMahon, "The Symbolics Genera Programming Environment", *IEEE Software*, Nov. 1987.