# A Diagram for Object-Oriented Programs

*Ward Cunningham*
*Kent Beck*

Computer Research Laboratory
Tektronix, Inc.

Abstract

We introduce a notation for diagramming the message sending dialogue that takes place between objects participating in an object- oriented computation. Our representation takes a global point of view which emphasizes the collaboration between objects implementing the behavior of individuals. We illustrate the diagram's usage with examples drawn from the Smalltalk-80™ virtual image. We also describe a mechanism for automatic construction of diagrams from Smalltalk code.

## 1. Introduction

The Smalltalk-80 virtual image [Goldberg 83] has many examples of expertly organized programs, many of which play an important role in the Smalltalk-80 system. These are worthy objects of study for at least two reasons. First, they often provide exquisite examples of the style and idiom unique to object-oriented programming. As valuable as a concise definition of object-oriented programming might be, it could not replace the corpus of code in the virtual image for illuminating the range of application of the object-oriented style. Students of object-oriented programming should be grateful that the implementors of Smalltalk pushed the object metaphor to the limit, building their system out of nothing but objects. Their result offers a guide to the "objectification" of even the most elusive algorithms. One learns by doing and one does Smalltalk by browsing the work of other programmers. Second, many Smalltalk objects are directly reusable—an even more compelling reason to study their behavior. To the degree that one's application mimics Smalltalk's own implementation, one will find useful objects in the image, preprogrammed, waiting to be used. Smalltalk's reputation as a user-interface-prototyping environment stems from its wealth of reusable user-interface components.

We sought a way of presenting the computations employed in Smalltalk's modular user interface. We developed a representation, a diagram, that emphasised the object nature of the interface components. The essence of an object-oriented computation is the dialog carried out by the objects involved. With this in mind, we consciously omitted from the diagram indications of state or sequence. So reduced, a single diagram shows the cascade of messages that result, for example, from a user interaction. The diagrams are not unique to user-interface codes, though, they are unique to object-oriented computations. We have since applied the diagramming technique to many of the more esoteric examples from the Smalltalk-80 image. The result has solidified our own understanding of object-oriented programming, and enabled us to teach others more clearly about Smalltalk-80.

In this paper we will introduce the notations of our diagramming technique and apply them to several examples from the Smalltalk-80 image. Later examples are drawn from behaviors in the image that are often misunderstood. We hope in this way to make a convincing demonstration of the diagrams' usefulness. Also, the reader can expect to see glimpses of the unique suitability of objects in implementing user-interfaces. We close with a discussion of an automatic technique for the construction and formatting of publication quality diagrams.



Smalltalk-80 is a trademark of the Xerox Corp. PostScript is a trademark of Adobe Systems, Inc.

## 2. The Diagram

We begin with objects. Objects accept responsibility for their own behavior. As a convenience, the code that implements this behavior is collected into a common place for all objects of the same class. Further, objects of one class might vary in behavior only slightly from those of another class. A new class is said to refine another if it implements only the variant behavior while relying on the other class for the remainder.

We represent an object as a box (See figure 1a.) A message sent to an object excites behavior specific to that object. We draw a message-send as a directed arc landing within the receiving object. If more than one object participates in a computation then there will be more than one box in the diagram (See figure 1b.) When one object invokes a computation in another object through a message send, we show that send as an arc originating in the sending object and landing in the receiving object. With the message goes a transfer of control. That is, the computation of the sender is suspended until the computation of the receiver is completed. Control returns backward along a message arc along with the answer to the message, if any. So far this mimics the usual semantics of procedure call. Note that we draw a particular message arc only once, even if the message is sent repeatedly, in a loop or otherwise.

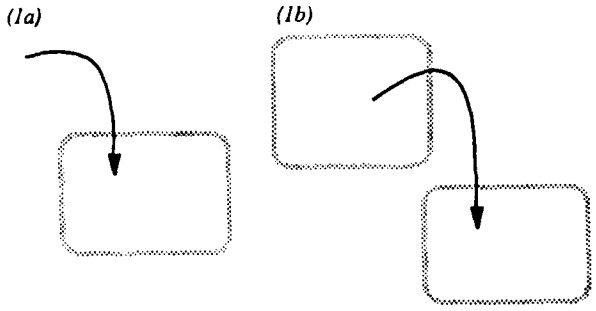


Figure 1. a) An object receiving a message. b) An object sending a message to another object.

An object will exhibit behavior appropriate for the specific message it receives. The various computations are implemented by distinct methods, each labeled with a method selector. We place the selector of methods invoked by messages at the receiving end of a message arc (See figure 2a.) It is important to note that the method invoked by a message will depend on the selector *and* the receiving object. The same selector might select different methods when received by different objects. In Figure 2b, for example, we cannot tell whether the two methods labeled "gamma" are the same. We need to know more about the objects involved.

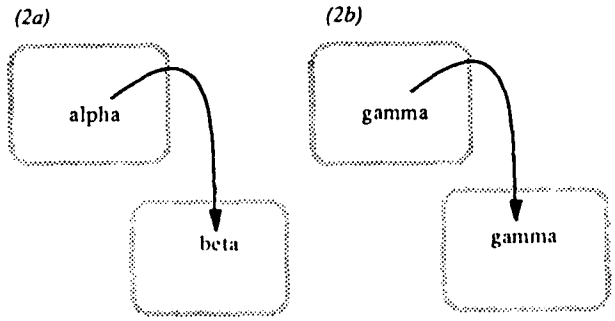


Figure 2. a) One method for "alpha" invokes another for "beta". b) One method for "gamma" invokes another for "gamma".

We identify an object in a diagram by its class. Recall that all members of a class share the same methods. The methods of the objects in Figure 3 are all exactly determined because we know the selector and the receiver's class for all of the messages. Recall also that objects of one class might inherit methods from another class. When methods are inherited from other classes (when a class does not implement a method, but one of its superclasses does) we divide the receiver into layers representing the classes involved and locate the method selector in the appropriate layer. Figure 3b shows two objects of two different classes (Senator and Plebe) each refining a third class (Citizen). The method for "gamma" invoked by each is in fact the same method, the one both inherit from Citizen. Of course, the same method won't necessarily execute in the same way in both cases; it is being executed on behalf of distinctly different objects. Figure 3c shows a revised Plebe. This time Plebes provide their own method for "gamma" which overrides the default implementation inherited by all Citizens.

We draw message arcs so that they always enter an object from above. When an arc travels across a layer of methods before finding its selector in a deeper layer, this suggests an opportunity to override that has not been exploited. The top layer will be that of the object's own class. Deeper layers will be superclasses. The bottom layer (if shown) will be the root of the hierarchy- class Object.

Note the contradictory use of the "elevation" methaphor by the terms "override" and "superclass". Which way is up? Some observers have complained that it is non-intuitive to place subclasses above superclasses in our characterizations of objects. We judge overriding the more important concept and like to think of method-lookup searching deeper for an implementation if none is provided by the surface class. Besides, we tried drawing the diagrams upside-down. They looked lifeless with their arcs limply dangling between method selectors.

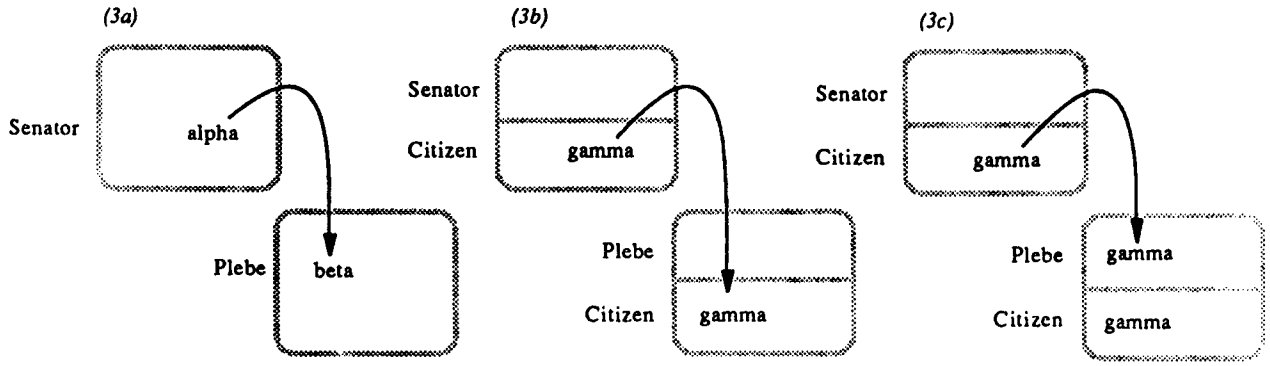


Figure 3. a) A Senator's method for "alpha" invokes a Plebe's method for "beta". b) A Senator's method for "gamma" invokes a Plebe's method for "gamma", in this case the same method inherited by all Citizens. c) A variant of b) where a Plebe overrides the inherited implementation of "gamma".

Consider an example drawn from the Smalltalk-80 image. The class Collection includes refinements for many commonly used aggregate data structures- arrays, sets, linked lists and the like. An OrderedCollection, for example, implements a flexibly sized array. An OrderedCollection responds to the message "add: anElement" by adding anElement at the next available location within itself. A slightly simplified diagram of this operation appears in Figure 4a. We can see that the add method makes use of two more elementary methods, size and at:put:. The diagram doesn't exactly explain why, but one could guess that size is used to determine where to put the new element and at:put: is used to put it there. Contrast this to the implementation of add: for Sets in figure 4b. This time the index is found by hashing the new element. Note that computing a hash function is the responsibility of the new element, not of the Set. All objects can hash themselves. Points, for example, compute their hash from their component x and y coordinates as illustrated in figure 4c.

We have now seen two examples of recursive behavior. In figure 3b the "gamma" method for one Citizen invoked itself for another. This style of recursion is common in Smalltalk especially when the objects are organized into a tree or list. In figure 4a we see a distinctly different kind of self reference. One method (add:) invokes others (size and at:put:) on behalf of the same object. This is done by addressing messages to "self". This is an idiom in Smalltalk since it is the mechanism by which complex methods are decomposed into simpler ones. Figure 5 illustrates some particularly interesting variations on this theme. The method "addAll:" works by adding each element of another Collection, one at a time. The algorithm works for all refinements that implement an appropriate method for "add:". We draw messages to self as arcs arching up and back down through the refining layers of an object, emphasizing the refinement's opportunity to override.

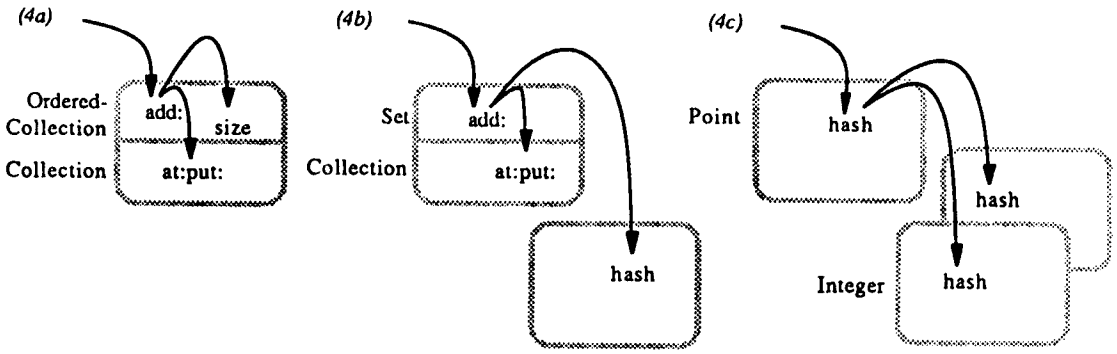


Figure 4. a) Adding to an OrderedCollection. b) Adding to a Set. c) Hashing a Point.

Smalltalk-80 provides a mechanism for a refinement to directly address methods of its superclasses. By addressing a message to "super" an overriding method can employ the method it is overriding as part of its implementation. We show a typical application in figure 5b. Note the absence of arch in this message arc. This visual distinction helps to make clear the difference in the way the method is found by the interpreter during a call to super, in contrast to the mechanism used in calls to self or other objects.

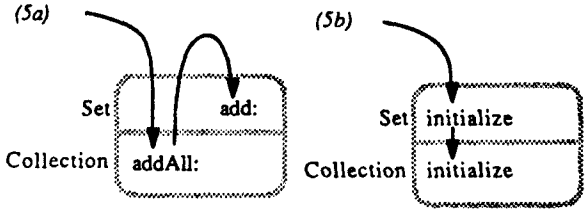


Figure 5. a) Adding all of one Collection to another. b) An initialization method invokes a similar method in a superclass.

### 3. Advanced examples

For more challenging examples we turn to the Smalltalk-80 user-interface. Smalltalk applications present themselves as windows on a bit-mapped display. A window may be divided into a number of panes, each displaying a different aspect of the application. By convention, keystrokes and mouse buttons are interpreted in the context of the pane touched by the cursor. Objects of class View and Controller accept responsibility for displaying output and interpreting input, respectively. A pair of objects, a View and a Controller, is allocated to each pane and another pair to the window as a whole. All of these are organized into a tree where the root represents the whole window, and the leaves represent individual panes. Finally, an object called the model, accessible to all of the Views and Controllers, represents the state and behavior of the application.

A window displays itself by recursively traversing its Views. Each View displays its border, its own contents, and any Views it might contain (see Figure 6a.) In practice, an application would employ refinements of View specialized for the needs of each particular pane. For example, a pane displaying a list might use a ListView that overrides displayView with the method for displaying lists. The actual contents of the list would be accquired from the model as shown in figure 6b. Note that the task of displaying a window has been decomposed using three separate programming techniques. First, several objects collaborate in the task. Second, the task is broken into parts for each object. Third, specialized objects can override any one of the parts. All of these techniques are visible in figure 6.

As a user interacts with an application's window, changes made to the model from one pane may require updates in others. The general mechanism for this is outlined in figure 7. A Controller recognizes user inputs as part of its controlActivity. When an input activity is complete, the Controller notifies the model that it has been changed. In response, the model notifies all of its registered dependents (Views always register themselves as a dependent of their model) of the need to update. The process of updating is left to the Views.

All Controllers cooperate to insure that the most appropriate Controller interprets the user's input at any given time. A Controller that wants control (because its View contains the current cursor point) gets control with the message "startUp". Figure 8 shows how this message eventually invokes the Controller's controlActivity. The controlLoop does

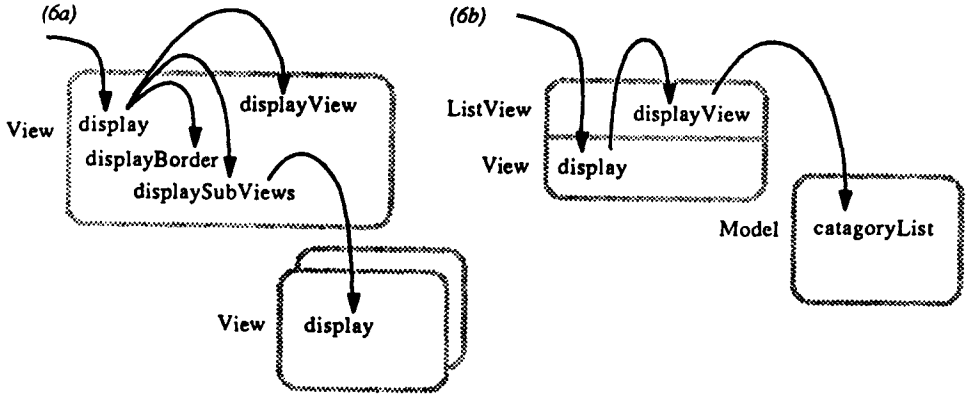


Figure 6. a) A View displays itself and its subviews. b) A specialized View displays a list accquired from its model.

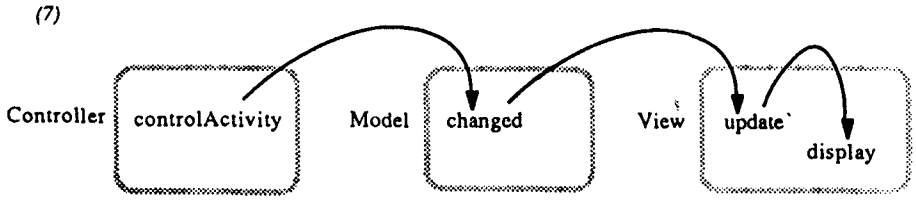


Figure 7. Views are advised of a change by a model.

the controlActivity repeatedly as long as the Controller remains active. In figure 8 we see the default implementation of controlActivity searching among its sub-Controllers for the appropriate controller for the moment. Refinements of Controller differ primarily in their implementation of controlActivity. Like the example of figure 6, this is simply a recursive traversal of the tree of panes. Both examples pass control to a critical method which can be overriden to implement specialized behaviors. Both will still inherit the behavior required to participate in the collaborative implementation of a user interface.

Comparing these diagrams with the Smalltalk-80 virtual image will reveal that we have bent the truth on many occasions. Yet, we argue, we have remained faithful to the style of the actual code. Our focus has been on the relationship between objects participating in a computation – a relationship that can be difficult to see when exploring objects one at a time. This is, in fact, the essence of object-oriented programming.

## 4. Creating Diagrams

Our notation emphasises the cooperation of objects participating in a computation. We freely omit portions of the computation judged unimportant. Such judgement comes easily enough when drawing a diagram by hand or with a general purpose drafting program as in figures 1 through 8. Our strategy for automating the drafting process had to admit intentional and asthetic considerations. Furthermore, we reasoned that only in the debugger [Goldberg 84], or more correctly, the simulation capability of the debugger, do the raw materials of the diagrams come together in one place. That is, to collect the information required for constructing diagrams we must do at least as much work as the debugger does when it steps a computation. The observation was fortuitous in that the debugger also had a user-interface that allowed one to step around computations judged uninteresting.

The Smalltalk-80 debugger is shown in figure 9a. It lists the activation records on the run-time stack, shows the source code of the selected activation, and provides two inspectors, one on the current object, the other on arguments and temporary variables. The debugger's menu has two commands to advance the computation, *step* and *send*. Step continues computing until the impending message returns a value. Send retains control so the user can watch the execution of the invoked method.

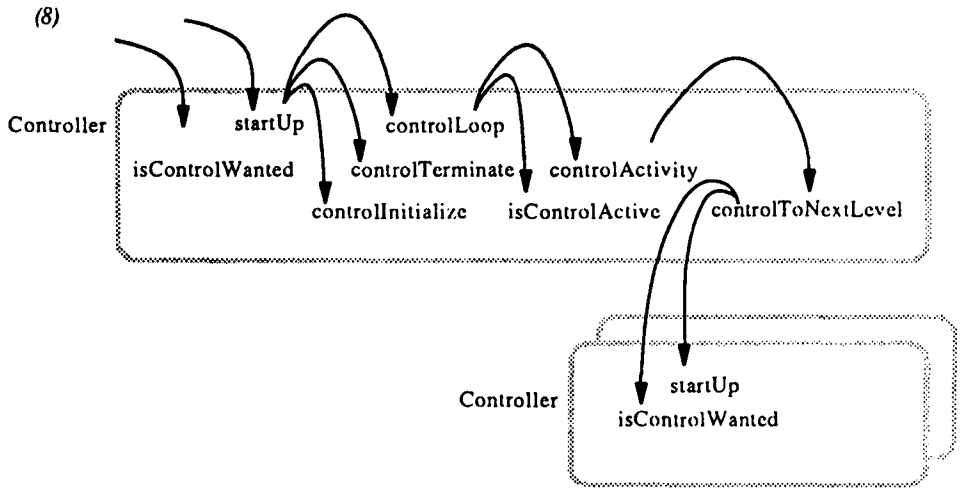


Figure 8. Control leads to controlActivity. The default controlActivity recursively passes control to a sub-Controller.

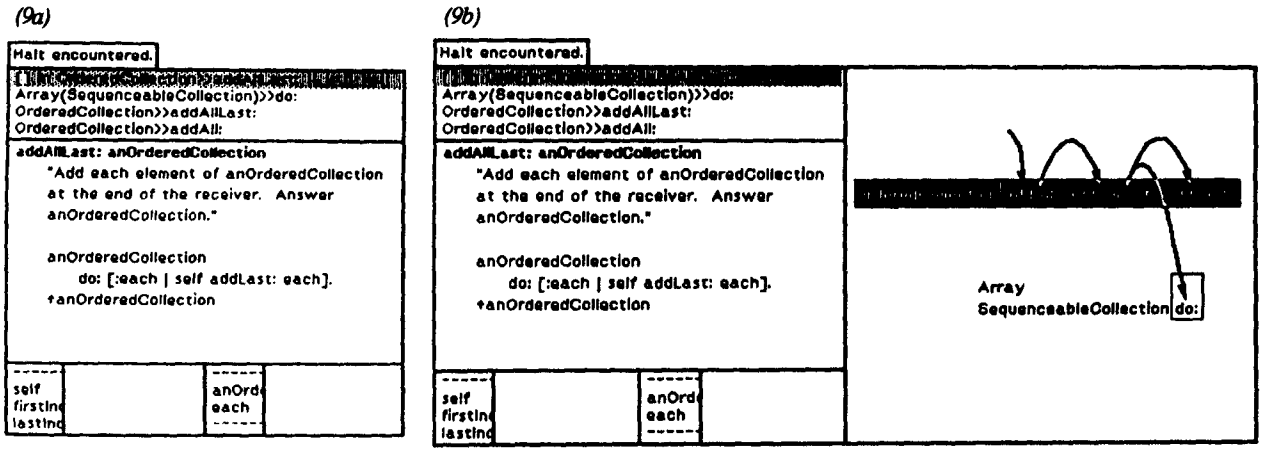


Figure 9. a) The Smalltalk debugger. b) The debugger with attached diagram editor.

We chose an extension of the debugger as our drafting user interface. The modified debugger is shown in figure 9b. The additional pane on the right is a special purpose diagram editor. The objects in the diagram can be moved around by dragging them with the mouse. Information is added to the diagram by menu commands *step* and *send*, which duplicate the ordinary debugger commands except that they record the message in the diagram. Objects and selectors are added to the diagram on demand. Objects require an initial placement by the user; selectors are positioned automatically.

To create a diagram one uses the step and send menu commands from the diagram pane to record the messages judged important. The original debugger commands can be used to locate the desired context without modifying the diagream. Other mouse operations are used to adjust the objects in the diagram until it is visually balanced and clearly conveys the computation.

A prepared diagram can be saved as a bitmap or as a high-resolution image encoded as PostScript™ [Adobe 85] function calls. The PostScript page description language provides a flexible way of specifying the contents of a typeset page. We wrote PostScript functions for each graphical element: arc, box, selector and class. A diagram is compactly encoded as a sequence of calls on these functions. Figures 10a and 10b contrast the bitmap and PostScript output.

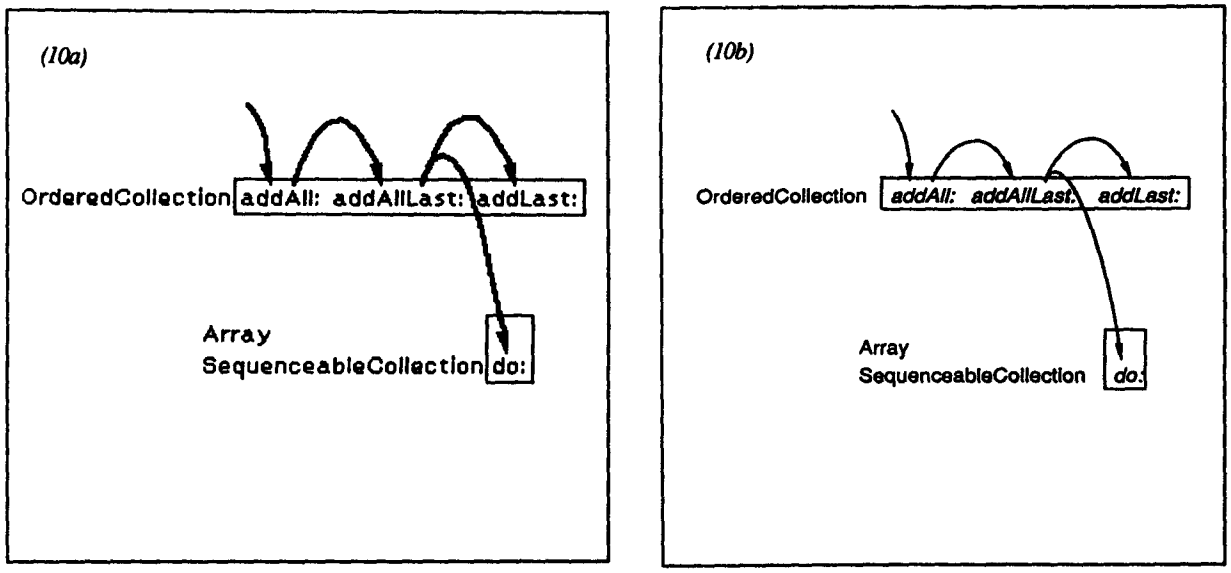


Figure 10. a) The bitmap output of the diagram editor. b) The same diagram with Postscript output.

We were struck by the dynamics of the diagrams as they developed on-screen, in the debugger. The objects in the diagram provided a helpful spatial reference to the objects involved in the computation. In this regard our work is similar to Brad Meyers visualizations in Incense [Meyers 85]. We found that by highlighting the currently executing object we were able to trace much more complicated computations than with the unmodified debugger. The facility proved helpful in locating a long standing bug in Smalltalk's coordinate transformation code. (The bug involved the inappropriate division of responsibilities between panes of a window.) Without the diagram, we had had trouble keeping track of which pane in the computations had done what when. Unlike the debugger's runtime stack display, which reflects only the current state of the computation, the extended debugger's diagram accumulated information throughout the debugging session.

As we pointed out early on, our notation does not explicitly represent the sequence of a computation. However, since the debugger obviously follows a sequence, and our diagrams accumulate in the debugger, our notation represents sequence when viewed over time. Inspired by related work in program animation by Ralph London and Rob Duisberg [London 85], we tried recording the dynamic behavior of a diagram as constructed. We hoped that this, played back at high speed, would add further insight into a computation. Playback involved first displaying the objects in the diagram, then, for each recorded message, drawing or redrawing the message arc and the receiving selector. Replaying diagrams has yet to substantially improve our understanding of a computation, probably because we have yet to view an animation that we haven't just finished recording. Replaying is sufficiently promising that we intend to try it in an instructional context using projection display equipment.

## 5. Conclusion

We have presented a way of diagramming object-oriented computations. Objects in a diagram are represented by boxes, labeled by the object's class and possibly its superclasses. The classes are listed with the most concrete class at the top, giving a natural interpretation to the term "overriding". Messages are represented by directed arcs from the sending object to the receiving object. Selector names at either end of an arc identify the sending and receiving methods. Furthermore, selector placement within an object indicates the class in which it is defined.

We have used these diagrams to teach beginning and advanced object-oriented programming to more than one hundred students. We feel that their use enhances our students' ability to understand some of the more esoteric examples in the Smalltalk-80 image. Those programs which rely on a dialog of several objects are much easier to understand with diagrams than just by examining source code. The user interface code, recognized to be some of the most difficult to understand, is particularly amenable to a diagrammatic treatment.

We have also extended the Smalltalk-80 system to automatically collect information for diagrams, and we have provided an editing and formating facility for the result. We implemented this as an extension to an existing utility, the debugger, to provide a familiar user interface. We have in the process enhanced the utility of the debugger.

In conclusion, we feel that the use of diagrams such as these can help teach the concepts and practice of object-oriented programming to naive users. In addition, we feel they can give more experienced programmers insight into how a complicated object-oriented system, such as the Smalltalk-80 virtual image, divides the responsibility of a computation. This insight is critical in a system like Smalltalk which relies on reuse to enhance programmer productivity.

## References


[Adobe 85] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, 1985.

[Goldberg 83] Goldberg, A. J., Robson, D. *Smalltalk-80: The language and its Implementation.* Addison-Wesley, 1983.

[Goldberg 84] Goldberg, A. J. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[London 85] London, R. L., Duisberg, R. A. Animating Programs Using Smalltalk. *IEEE Computer* 18(8):61-71, Aug 1985.

[Myers 85] Myers, B. A. Incense: A System for Displaying Data Structures. *Computer Graphics: SIGGRAPH '85 Conference Proceedings,* pp 115-125. ACM, July 1985.