Published in Visual Object Oriented Programming, M. Burnett, A. Goldberg, & T. Lewis, eds., Prentice-Hall, 1995, pp. 185-198

Getting Close to Objects: Object-Focused Programming Environments

Bay-Wei Chang
Computer Systems Laboratory
Stanford University
Stanford, CA 94305
bay@self.stanford.edu

David Ungar
Sun Microsystems Labs, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
david.ungar@sun.com

Randall B. Smith
Sun Microsystems Labs, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
randy.smith@sun.com

ABSTRACT

Current visual programming environments make use of views and tools to present objects. These *view-focused* environments provide great functionality at the expense of distancing the objects behind the intermediary layers of views and tools. We propose the *object-focused* model, which attempts to foster the notion that objects themselves are directly available for interaction. Unique, directly manipulable representations of objects make them immediate, and basing functionality on the object rather than on extrinsic tools makes them the primary loci of action. But although immediacy and primacy contribute to the sense of concreteness of the objects, discarding conventional views and tools potentially restrict the functionality of the environment. Fortunately, by being extremely faithful to the notion of concreteness of objects, two principles emerge that allow object-focused environments to match the functionality of view-focused environments. The principle of availability makes functionality of objects accessible across contexts, and the principle of liveliness allows objects to participate in multiple contexts while retaining concreteness. All these elements help make objects seem more real in the object-focused environment, hopefully lessening some of the cognitive burden of programming by reducing the distance between the programmer's mental model of objects and the environment's representation of objects. Programmers can get the sense that the objects on the screen *are* the objects in the program, and thus can think about working with objects rather than manipulating the environment.

1 INTRODUCTION

Visual programming's attractiveness stems in large part from its immediacy—programmers directly interact with program elements as if they were physical objects. These concrete visualizations of the program on the computer screen shape how programmers visualize the program within their mind, and may give them a foothold from which to think about the program: people find it easier to deal with the concrete than with the abstract. Object-oriented programming languages (even those that rely primarily on textual representations) strive toward the same end by providing objects as the fundamental elements in the program. Objects encapsulate and make concrete the elements of the program, including both the data to be manipulated and the behavior to be applied to them. Again, this paradigm works because most people find it easier to deal with the concrete than with the abstract.

Yet while object-oriented programming languages assist the programmer by providing a concrete notion of objects, most programming environments for these languages push the programmer in the opposite direction, back toward the abstraction of objects. Rather than presenting objects directly, the environments present intermediaries that show certain aspects of the object—for example, an inspector shows instance variables but cannot show all references to the object without launching another tool. This approach has the effect either of fragmenting the object into many different aspects, weakening the sense of a single unified object, or of distancing the object deep in the recesses of the computer, making it reachable only through intermediaries acting to show aspects of it.

Our premise is that a visual programming environment that closely matches the programming language can lessen the cognitive load of programming, by reducing the distance from the programmer's model of the object to the programming environment's model of it. We propose that programming environments employ the principles of *immediacy* and *primacy* to present objects as concrete entities that are the primary loci of action, rather than as abstract entities which are secondary to the tools and views used to examine and manipulate them.

But by eschewing conventional views and tools in favor of direct interaction with objects, it may seem that we must also give up the functionality that views and tools provide. For example, views show multiple representations of the object, allow separate instances of the object on the screen, and are gathered together into useful tools, like browsers and debuggers. However, by taking the idea of independent, individual objects very seriously, we have discovered two principles, *availability* and *liveliness*, which enable us to attain similar levels of functionality in a concrete world.

We begin by discussing the differences between conventional view-focused environments and the proposed object-focused paradigm, concentrating on the nature and identity of objects in those environments. The Seity interface, an

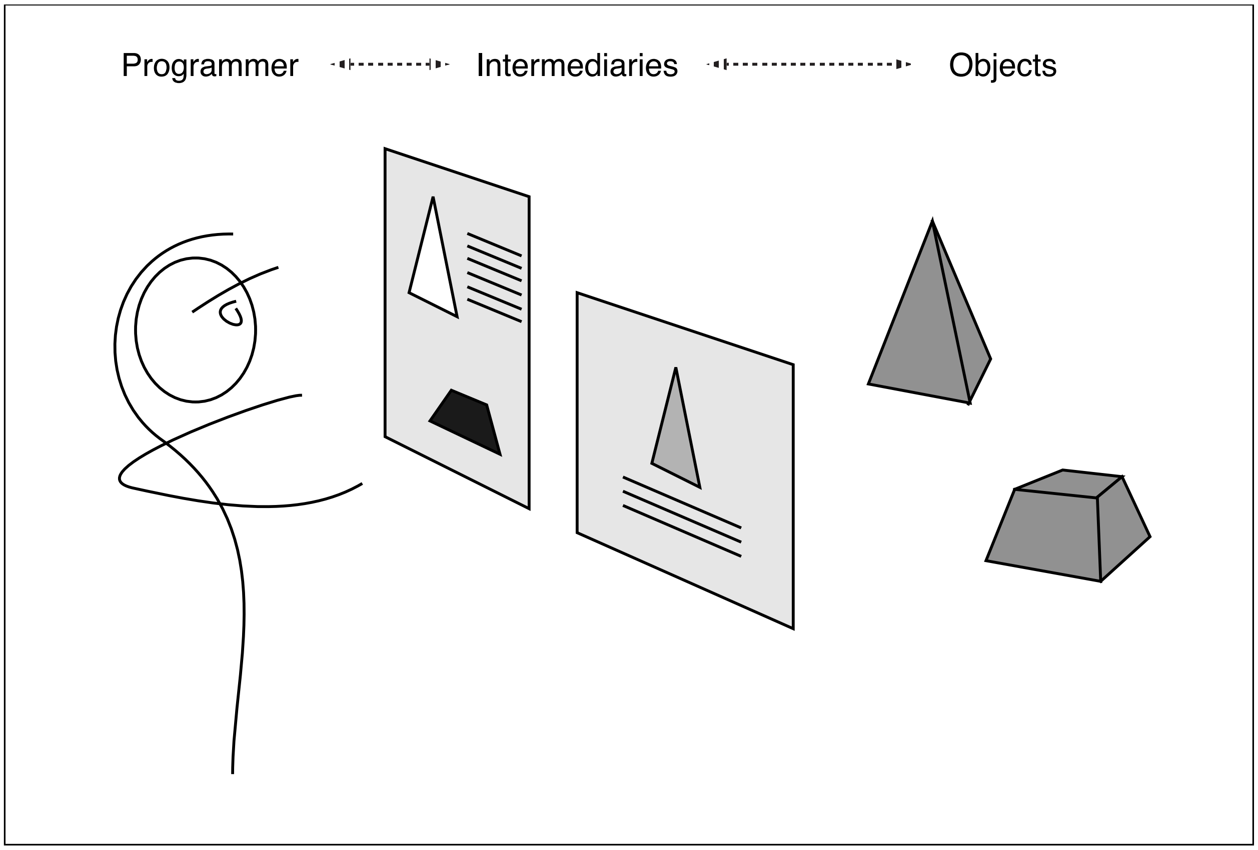


Figure 1. In a view-focused environment, the programmer interacts with intermediaries (tools), which present views of the objects.

experimental browsing user interface for the programming language Self, is used as an example of a visual environment that focuses on objects rather than on views and tools. Within Seity, Self code is still represented textually, but the objects of the language are presented directly as concrete entities.

2 VIEW-FOCUSED VERSUS OBJECT-FOCUSED ENVIRONMENTS

Most current programming environments for object-oriented languages are *view-focused*: objects are examined and manipulated through intermediaries, each of which permit a certain view of the objects (Figure 1).

The Smalltalk-80 environment [Goldberg 1984] is the seminal object-oriented exploratory programming environment. Tools like class browsers, method browsers, protocol browsers, and inspectors provide various perspectives of the objects in the system. Tools are associated with objects, but the same object may be shown by many tools, and the same tool may show different objects over its lifetime. The tools are concrete—they are the things that are manipulated in the environment—but, in a formal sense at least, the objects are abstracted, distanced, and secondary. The objects are abstracted because they reveal different parts of themselves within multiple, separate forums (the tools) and because they maintain no central locus of existence or manifest identity. The objects are distanced because they cannot be reached directly by the programmer, but must be viewed

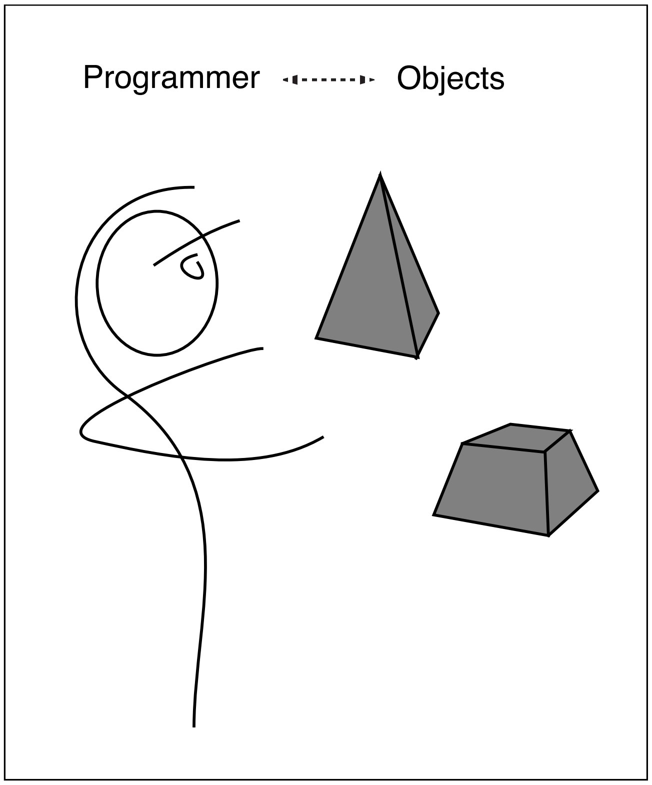


Figure 2. The object-focused environment fosters the notion that the programmer is interacting directly with the objects.

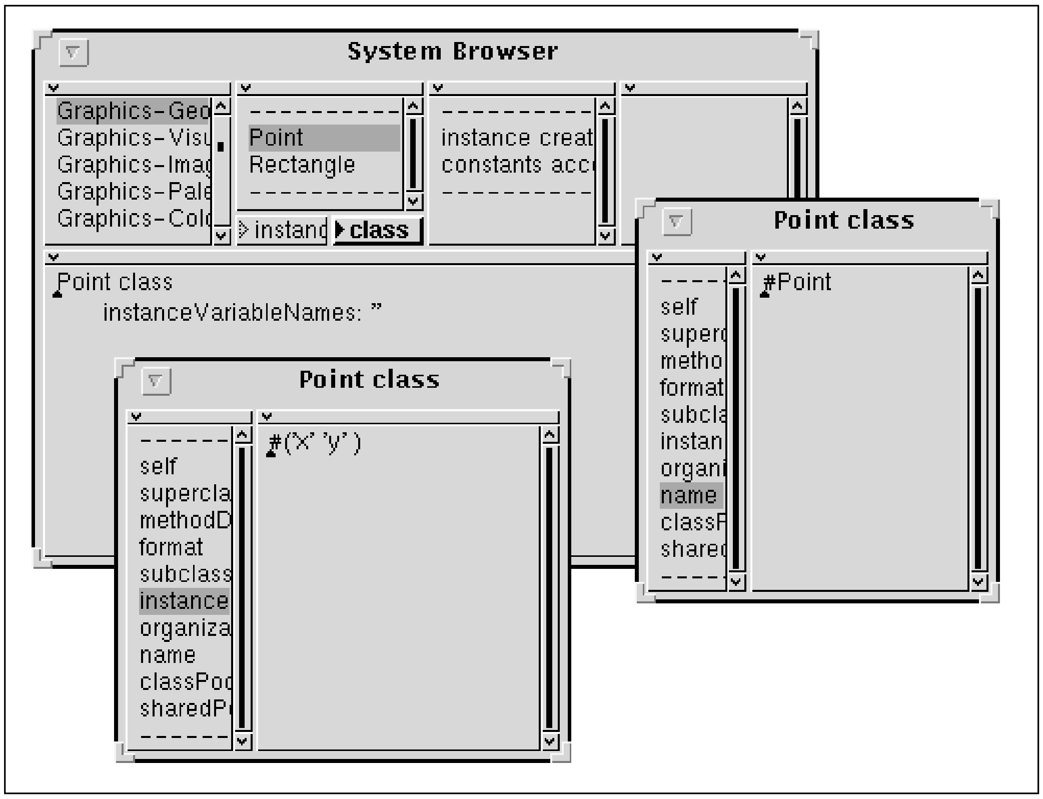


Figure 3. The same object (in this case, the Point class) can show up in multiple tools in the Smalltalk environment; the same system browser can view many different classes and methods.

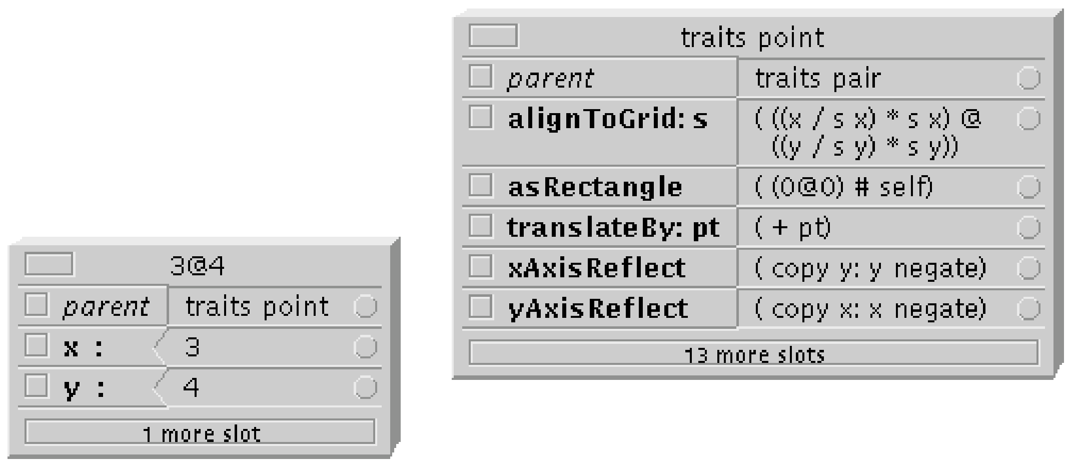
through tools. The objects are secondary because the tools are manipulated as the primary action (Figure 3).

However, this is the most formal interpretation of tools in a view-focused environment. The programmer can come to identify tools with the object, as the tools become transparent with frequent use. Thus a Smalltalk inspector is thought of as the object which it inspects, thereby pushing the object to the forefront and eliminating the distance to it. The *object-focused* model takes this concept as a starting point, attempting to eliminate the sense of an intermediary from the first (Figure 2). The object in the interface is intended to be concrete, immediate, and primary. For example, Seity [Chang 1994], an experimental user interface for the prototype-based object-oriented programming language Self [Ungar and Smith 1987], presents a Self object as a box with a column of slots on its face (Figure 4).

Seity currently represents all Self objects in the same way. This is consistent with the Self object model: Self is prototype-based, meaning it has no classes. New objects are created by cloning existing objects; shared behavior (for example, methods that would reside in the class for a class-based language) is inherited from other objects. The uniformity of this model encourages a uniform way of looking at any object, hence the basic set-of-slots representation in Seity. However, the object-focused model is not only applicable to prototype-based languages. There is no reason why class-based languages like Smalltalk and C++ cannot be supported by an object-focused programming environment. While in Self there is only one kind of object, in Smalltalk there are instances, classes, and metaclasses. These objects would appear in the environment with different representations, each showing appropriate information about itself. The non-object constructs in languages like C++ would also appear in the environment with identities appropriate to their function. The important thing is not that a particular object model exists for the language (a non-object-oriented language could also be represented), but that the language be divided into logical units to be mapped to on-screen objects. When the division is a useful way of conceptualizing the program elements, the units will be able to function effectively in the object-focused programming environment. There will be no need for other windows, browsers, or other such tools with which to view objects, because the programmer can already see and interact with the objects themselves.

But what does the phrase “the objects themselves” actually mean? After all, the object is simply a mutually agreed notion between the programmer and the computer, based on the programming language. This raises two issues: first, whether a concrete representation can be chosen for the object that not only can stand for the object, but in fact can come to be accepted as the object itself (the principle of *immediacy*); and second, what that representation should be.

In the object-focused model the on-screen representation of the object is considered to represent the object itself, not merely a singular tool through which the object shows itself. To this end, there is never more than one such representation of any given object (though this representation can be mal-



3@4	
parent	traits point
x :	3
y :	4
1 more slot	

traits point	
parent	traits pair
alignToGrid: s	(((x / s x) * s x) @ ((y / s y) * s y))
asRectangle	((0@0) # self)
translateBy: pt	(+ pt)
xAxisReflect	(copy y: y negate)
yAxisReflect	(copy x: x negate)
13 more slots	

Figure 4. Two Self objects in Seity.

leable). In addition, this particular on-screen representation is never associated with any other object. Since there is a one-to-one mapping between the representation and the object, it is a small step to discard the strictly accurate notion that the representation is an intermediary separate from the object, and to consider the concrete representation to actually *be* the object. While the programmer surely does not truly believe that the on-screen representation is the object, it is convenient and natural to act as if it were: the offered concrete manifestation is easier to embrace than an abstract one, and the object is at hand to be examined and manipulated. This identification encourages programmers to feel as if they are directly dealing with the objects of the language. Hutchins et al. [Hutchins et al. 1986] refer to this sense of immediacy with the semantic objects of interest as *direct engagement*. Direct engagement is achieved in view-focused environments when tools become transparent; by eliminating the levels of indirection introduced by view-focused tools, object-focused environments may increase the frequency of direct engagement.

The constraint in choosing the representation itself is that it must match the model of objects in the language. An object in Seity, for example, is presented essentially as a set of name-value pairs, which corresponds precisely with a Self object. This of course leaves wide latitude in the details of the representation—the Seity representation could have as easily been a simple rectangle instead of a 3D slab. But these details are important, for the programmer will come to identify the representation with the language object, and it will shape his or her mental model of objects. The concrete representation informs the programmer of the nature of the objects, and the more accurately it can reflect a useful model of them, the more likely the programmer will feel comfortable working with such representations in the environment.

The concreteness of Self objects is heightened in Seity by making it seem as if those objects are physical, real-world things. Thus their concreteness is not only one of spatial localization and identity, but also one of image and behavior: the objects look like 3D slabs, and they move solidly. An object has an identity; it never appears more than once on the screen. Instead, multiple references to it refer (via arrows) to the same instance (Figure 5). Furthermore, the objects are placed in an environment that can be called an artificial reality, a world that has some features reminiscent of the real world but is primarily a consistent, individual universe unto itself. In the Seity Self reality, objects move smoothly with the use of animation, employing cartoon-inspired techniques to make their movements seem lively, engaging, and realistic [Chang and Ungar 1993]. The use of visual solidity and animation helps make objects seem real, furthering the illusion that these are indeed the objects in the program.

The overall effect of making objects immediate is, in fact, to make them seem real. Our goal is to create the sense that the object is directly available to the programmer: the object is the thing right there on the screen.

3 OBJECTS AT THE CENTER OF ACTION

Of course, just seeing the object isn't enough to give it a complete sense of identity. It must also have appropriate properties associated with it, just as real-world objects have properties. That is, just as the properties of real-world objects define, individualize, and unify them, the properties and behavior of objects in the programming environment ought to do the same. Since its properties and behavior are properly owned by the object, the programmer must go to the object to get at them—the principle of *primacy*. Consolidating the functionality needed of the object within itself serves to centralize focus on the object, rather than on outside view-focused tools that have traditionally encapsulated areas of functionality. Again, this makes the objects primary in the environment, which in turn brings the programmer closer to the objects. But can all the functionality of traditional view-focused tools be preserved in the object-focused model?

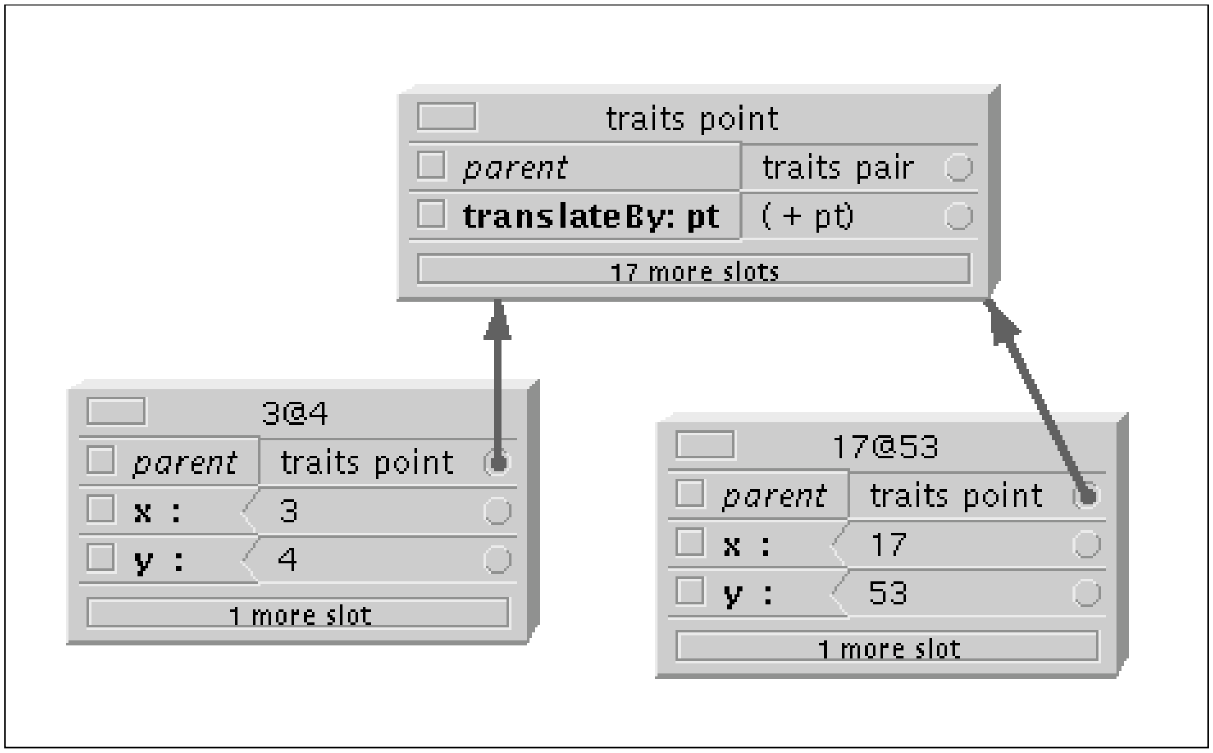


Figure 5. Objects in Seity maintain their identity by not appearing more than once on the screen.

3.1 VIEWS AND TOOLS REVISITED

For the purposes of this discussion, the term *view* will be used to mean a particular way of presenting some subset of information within an object. Views allow an objects to be understood from defined perspectives, carving up the large semantic space of the object and packaging up the chunks in useful and comprehensible configurations. They specialize the presentation of the object for a particular task—for example, showing the values of variables, the code of methods, or the place on a stack. Each view is a discrete unit of functionality.

Tools, as used in this discussion, are objects in the interface that centralize functionality for a given activity. They are a conglomeration of views, with ways to manipulate those views. A debugger tool, for instance, might show the current stack, with ways to look at activations on that stack and objects involved in those activations. Several views of different objects would be involved in a single debugger tool. What the views have in common is that the functionality they provide is needed in this one activity: debugging.

View-focused tools for programming emphasize the activities the programmer engages in. There are tools for browsing objects, for writing code, for debugging, and for presenting inheritance hierarchies. View-focused environments make the programming activity explicit, but at the expense of somewhat distancing the targets of that programming activity, the objects with which one is working. (This has also been called the activity-focused model [Hedin and Magnusson 1988].) The objects are somewhat subordinated to the operations on them and the ways of looking at them.

3.2 OBJECT-FOCUSED FUNCTIONALITY

Object-focused environments achieve functionality by giving objects the behavior needed for studying and changing them. Properties of the object are part of the object, so we just need to know how to get at them—perhaps by pressing a button, choosing a menu item, or selecting an option from a pull-out drawer. The same kind of direct manipulation techniques used to manipulate view-focused tools can be applied to manipulate the objects themselves. The difference is that the result affects the object itself, not a disconnected intermediary. The object may show more information, change aspects of its presentation, or even radically transform its representation. In each manifestation, however, it is clear that the object itself is being manipulated and examined.

A Self object in Seity, for example, is a complete entity. No separate browser or inspector is needed to examine it—the object itself serves that purpose. Poking a button gets the contents of a slot (the object in the slot grows from a dot within the button to a full-sized 3D slab, see Figure 6). Poking other buttons can remove the object from the screen (it falls off the bottom of the screen), or hide a slot from view (the slot slides out from the slab and get sucked into the bottom of the slab). Other actions, like finding all the references to the object, are initiated from menus integrated into the object (Figure 7).

While view-focused environments make the programming activity explicit, the object-focused model incorporates the manipulation of the objects as the general activity; programming activities are implicit within those manipulations. Rather than centralizing functionality into a monolithic tool, smaller chunks of functionality are coupled to the object, so

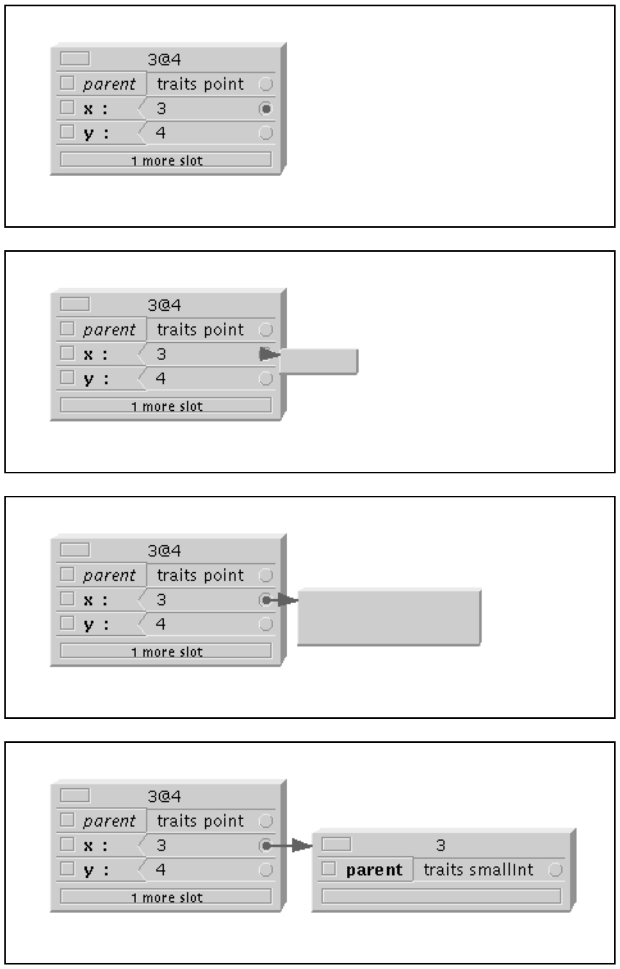


Figure 6. Seity allows examination of the contents of a slot by poking directly on the slot to get the object inside.

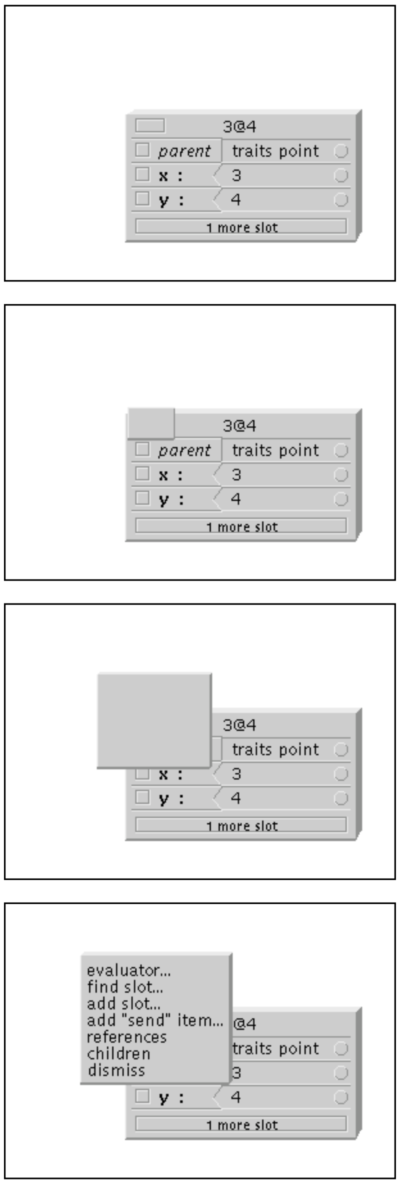


Figure 7. Functions not available directly on the object are provided in menus which present themselves as part of the object.

that any functionality can be invoked at any time on the object, not only when it is being viewed by a particular kind of tool. This principle of *availability* frees the programmer from modes in which limited portions of object functionality are available at any one time—for example, within a view-focused debugger tool it might be possible to inspect an object's slots, but it might not be possible to find all references to that object without going to a separate tool. In addition, object-focused functionality naturally shares functionality across conceptual domains of activity. The activities of debugging, browsing, and creating objects might all require inspecting the contents of slots, or finding all senders of a message. Rather than distribute this functionality of the object in many places, as discrete tools would, objects in the programming environment implement it as part of themselves. When objects are concrete, their behavior is always available. The principle of availability is a natural consequence of the object-focused model.

3.3 VIEW FUNCTIONALITY FOR CONCRETE OBJECTS

Clear benefits accrue from the ability to view an object in more than one respect, and it would be a loss if they were sacrificed on the altar of concreteness. Luckily, concrete objects in the object-focused model can indeed have multiple views—by transforming themselves into different visual representations when requested. Like a view in a view-focused tool, the object is specialized for the current task; but unlike the view-focused tool, it is implicit that the thing on the screen is still the same object, just showing a different face. The concreteness is retained because the view is not decoupled from the object.

Multiple simultaneous views, which are useful for comparing disjunct aspects of the object at the same time, can also be handled. Objects show the two representations at once, but in keeping with the goal of concreteness, the object remains connected and solid. When multiple simultaneous views of an object are needed in separated locations, the object-focused model potentially runs into trouble. The essence of the object-focused model is that objects have a single identity. The single strongest cue that objects are concrete is that, like real-world objects, they cannot be in two places at one time. Disjoint instances of the same object would destroy the illusion of the identity of the object, weakening the sense that the things on the screen are the objects in the program. This functionality could still be provided by reifying the remote views as view objects, but there may be a better way.

We suggest that the principle of *liveliness* can at once provide much of the functionality of disjoint views while increasing the sense of concreteness of objects. A *lively* interface allows objects to move, change, and interact under their own power. The programmer still controls what happens, but the interface is no longer a static environment, passive until the programmer reaches out and grabs an object or pushes a button. Objects go where they are needed, when they are needed.

We have experimented with liveliness in the Seity interface: objects enter and leave by moving on and off screen under their own power; self-dismissing notifiers simply drop off the screen if they have not been handled; and an object can be arranged to come to the cursor on the click of a mouse button (Figure 8).

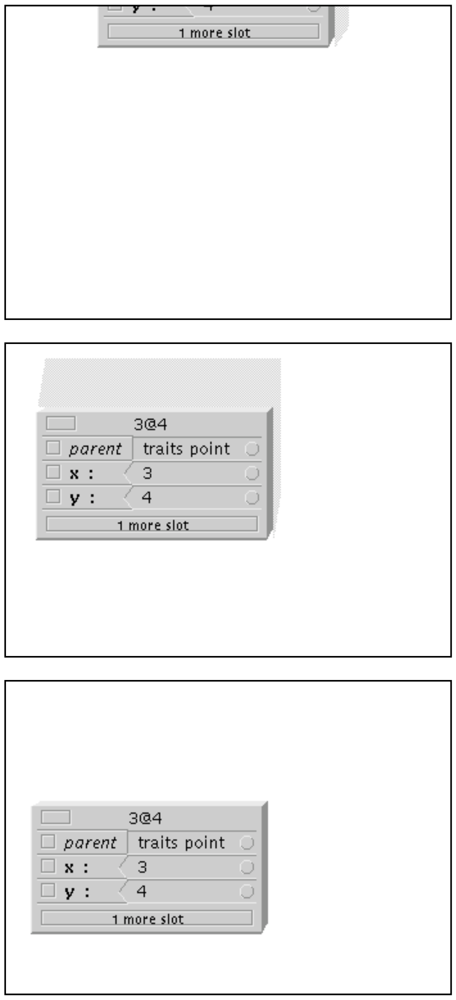


Figure 8. A simple example of liveliness in Seity: objects enter the screen under their own power.

Liveliness mitigates the need for disjoint views by letting objects move from one place to another and transform themselves along the way if necessary. If an object is needed in multiple places, it can move to those places when it is needed, depending on the programmer's desire or the environment's needs. The objects are in collaboration, not competition, with the programmer.

3.4 TOOLS—COOPERATING CONCRETE OBJECTS

Tools, in the traditional view-focused environment, bring together multiple objects in order to operate on several at once, or to show relationships among them. The object-focused environment needs the ability to work with multiple objects at once as well. In fact, we need not give up the formal notion of a tool, if we recast it as an organizer and coordinator of multiple objects, rather than as a repository for remote views of objects. The role of the coordinator is then played by an object intimately connected with the activity, not an outside presence.

For example, consider the activity of debugging. An object-focused environment can present the process object, which has a button that will show the current stack. Pressing the button begins assembling the stack, which calls in the activation objects. The objects may simply gather in a column, connected to one another by arrows, or the process object might sprout a shelf and the activations will align themselves in a tower on that surface. Just like in a traditional view of a stack, the result is a list of activation frames, but the frames feel more like the activation objects themselves than they might in a traditional view. Because they are *lively*, we can pull activations off the stack, confident that when we want to look at the stack as a whole again, the unglued activations will remember their role and return. Because they are *available*, any browsing, searching, of modifying—any functionality ever available in the object—can be performed on the activation objects without going elsewhere or changing modes. This debugging "tool" is merely the conglomeration of lively, available, concrete objects cooperating under the direction of another, the process object. It can be seen as a whole unit, or as a momentary collaboration between many different objects. The participating objects are free to be coaxed out and individually manipulated at any time, and can even participate in more than one activity at one time, by shuttling themselves around when necessary.

Cooperating concrete objects can mimic any view-focused tool to provide a locus in which several objects are acted on or show their relationships to one another. But concreteness increases the utility of this locus over that of a view-focused tool by maintaining the individuality and usefulness of each participating object.

Currently, Seity does not have facilities for multiple views of objects or object-focused debugging such as that described above. However, our experience with the object-focused browsing environment suggests that a full object-focused programming environment could be both effective and pleasing to use.

4 SUMMARY

Object-oriented programming languages present an opportunity to break out of the largely text-oriented environments of conventional languages. We propose the object-focused model of an object-oriented programming environment, in contrast to the traditional view-focused model. In the object-focused model, objects are made concrete by enforcing their unique identities; they are made immediate by showing directly manipulable representations unfettered by intermediaries; and they are made primary as the source of action by basing functionality on the objects rather than on remote tools. In addition, the object-focused model achieves most of the functionality natural to a view-focused environment by applying the principle of availability, which makes available all the functionality of objects in any context, and the principle of liveliness, which allows them to move and change under their own power.

The resulting object-focused programming environment can make objects seem more real, potentially lessening some of the cognitive burden of programming by reducing the distance between the programmer's mental model of objects and the environment's representation of them. Programmers can get the sense that the objects on the screen *are* the objects in the program, and thus can think about working with objects rather than about manipulating the environment.

REFERENCES

1. [Chang and Ungar 1993] B. Chang and D. Ungar, "Animation: From cartoons to the user interface," *UIST'93 Conference Proceedings*, Atlanta, Georgia, Nov. 3-5, 1993, pp. 45-55.

2. [Chang 1994] B. Chang, *Seity: Object-Focused Interaction in the Self User Interface*, Ph.D. dissertation in preparation, Stanford University, 1994.

3. [Goldberg 1984] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1984.

4. [Hedin and Magnusson 1988] G. Hedin and B. Magnusson, "The Mjølner environment: Direct interaction with abstractions," *ECOOP'88 Conference Proceedings*, published as *Lecture Notes in Computer Science #322*, Springer-Verlag, New York, 1988, pp. 41-54.

5. [Hutchins et al. 1986] E.L. Hutchins, J.D. Hollan, and D.A. Norman, "Direct manipulation interfaces," in *User Centered System Design* (D. Norman and S. Draper, eds.), Lawrence Erlbaum, Hillsdale, New Jersey, 1986, pp. 87-124.

6. [Ungar and Smith 1987] D. Ungar and R. Smith, "Self: The power of simplicity," *OOPSLA'87 Conference Proceedings*, published as *SIGPLAN Notices*, Vol. 22, No. 2, 1987, pp. 227-241.