Debugging and the Experience of Immediacy

Article *in* Communications of the ACM · April 1997

DOI: 10.1145/248448.248457 · Source: DBLP

3 authors, including:

David Ungar
IBM
105 PUBLICATIONS **5,397** CITATIONS

SEE PROFILE

David Ungar, Henry Lieberman, and Christopher Fry

Debugging and the Experience o

A GOOD USER INTERFACE BRINGS YOU, THE USER, face-to-face with whatever is being manipulated and experienced. For example, the steering system in a sports car, especially one with a mid-engine layout, lets you feel the road surface so you can tell when the road is slippery just by the feel of the steering. A good tool, like a high-quality wrench, becomes part of your hand, so you feel your hand is turning the bolt directly. The machinery disappears, and you feel connected to the object in question. A programming environment can also convey the experience of immediacy, drawing the programmer closer to the program. When that happens, debugging is easier. The principle of immediacy can serve as a guide, keeping builders of programming environments on the path to productive environments.

Like your favorite pizza, a programming environment can be described as a layered yet synergistic stack of ingredients:

- Language semantics define how the user specifies a computation.
- Affordances define how the user examines, changes, and runs programs.
- A user interface brings the affordances into the user's sphere of perception and action.

The principle of immediacy can be brought to bear on each of these layers. For example, in the user interface, it helps if each chunk of information (e.g., each procedure) shows up as a visually distinguished and clearly legible unit. At the low level of visual perception, designers frequently employ cues that

When programming environments convey the experience of immediacy, programmers perceive and manipulate the facets of the program and its unfolding computation with less conscious effort—and accelerate the debugging process.

f Immediacy

mimic the real visual world in order to exploit a user's preexisting perceptual abilities. That is why, for example, the Self user interface [1] renders each object as a slab with stereotypical stage lighting, from above and left. Careful attention to perceptual legibility can ease the burden on the programmer, so more attention can go to finding bugs and less attention to finding objects on the screen. The user interface must allow programmers to manipulate programs as well as see them. Here again, immediacy can be achieved by employing affordances that match the user's prior experiences or that reduce the need for the user to change focus. A bad user interface draws so much attention to itself it distracts from debugging, and a good one lets a tired programmer debug much later into the night (if need be).

But the user interface is only the lowest level of the system. The programming environment provides the lenses and levers that let the programmer inspect, manipulate, and debug the program. But if the lenses and levers feel like lenses and levers, the programmer is distanced from the program, has to mentally work harder to keep track of it, and is encumbered in attempts to fix it.

The principle of immediacy leads designers of programming environments to understand that the environment should make it effortless to examine a program—with all of its connections. If the programming environment you use cannot rapidly and effortlessly show you all the callers of a function, all the definitions of a message, or all the places that can change a particular variable, it is distancing you from the logical structure of your program, making it that much harder for you to debug it.

When debugging, what goes for a static program also goes for a (running or suspended) process. When you find an argument with an erroneous value, the environment should be able to take you to where the value was ultimately passed in, even if it is 10 stack frames up (although we know of none doing this today). Immediacy means the important relationships are made manifest.

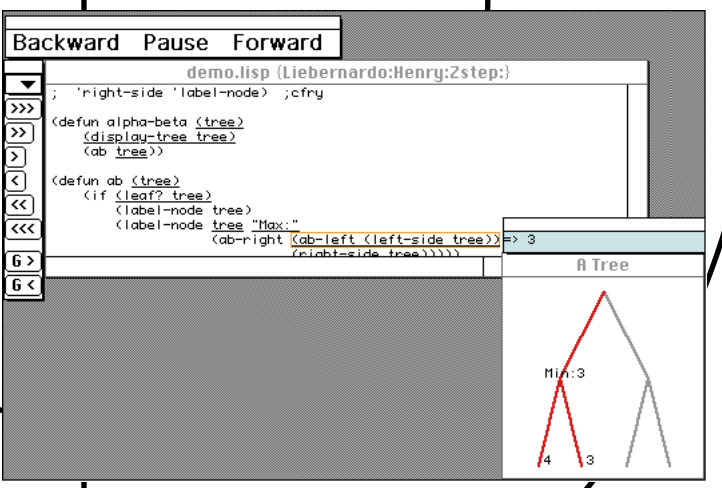

Figure 1. The ZStep 95 interface

Immediacy and Debugging

We illustrate how the principle of immediacy applies

to debugging environments by describing ZStep 95 [2, 3], a program debugging environment based on the principle of immediacy (see Figure 1). Debugging is often a search that starts from the observed effects of running a program and proceeds to deduce the possible causes in the underlying program code. So ZStep 95 strives to facilitate debugging by bringing the relationship between cause and effect closer to the programmer.

The separation between cause and effect can occur in at least three dimensions: time, space, and semantics. Thus, designers should strive for three kinds of immediacy important for debugging: temporal, spatial, and semantic.

A programming environment that cannot rapidly and effortlessly show all the callers of a function, all the definitions of a message, or all the places that can change a particular variable distances the programmer from its logical structure.

Temporal Immediacy. Human beings recognize causality without conscious effort only when the time between causally related events is kept to a minimum. If there is a delay between a change in a steering wheel and the response of a car, we describe the steering as "mushy" and it destroys the feeling of immediacy of control of the car. In programming, delay between an effect and observing related events or data in the program puts a strain on programmers' short-term memory to hold all the relevant information in their heads while waiting for the programming environment to catch up. In Eisenstadt's article on bug war stories in this issue, 15% of bugs are directly attributable to temporal distance between cause and effect—the largest single source of error in his survey.

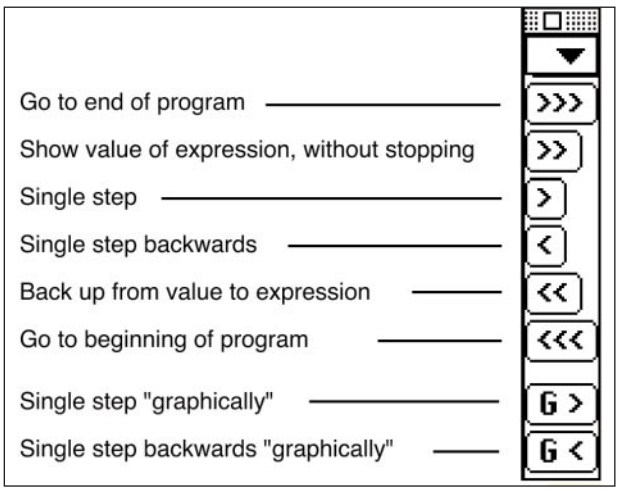

Figure 2. ZStep 95's "video recorder" controls

In debugging, it is important to reason backward from effects to causes, so temporal immediacy backward in time is just as important as, if not more important than, temporal immediacy in the forward direction. ZStep 95 is built on the notion of reversibility. It keeps a history of the computation and can be run either forward or backward using "video recorder" controls (see Figure 2). The history includes expressions, values, and the graphic output of the program. Also important is fine control over the level of detail displayed. It is not only immediacy between temporally adjacent events that is important but also immediacy between temporally relevant events. ZStep 95 provides both "fast" and "slow" stepping speeds in both directions, as well as manual and automatic control. By allowing the programmer to choose speed and direction, ZStep makes it easier to find a mode in which the tempo-

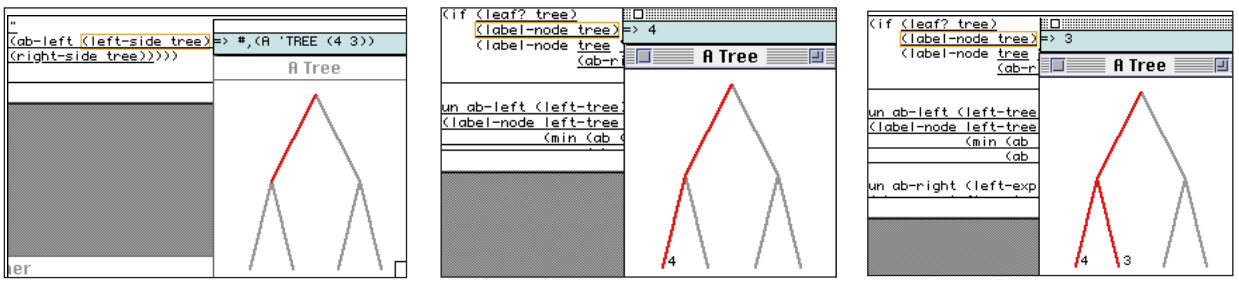

Figure 3. Three successive "graphic steps" of a tree-searching program

ral distance between relevant causal connections occurs on the appropriate scale for preconscious perception.

ZStep also provides novel `Graphic Step Forward` and `Graphic Step Backward` commands that work in terms of events in which something is drawn on the screen, rather than in terms of expressions in the code. These commands allow the user to step the behavior of the program rather than step the code and support temporal immediacy between graphic events and their causes in the code.

Whether the choice is to step expression-by-expression, graphic-event-by-graphic-event, forward, or backward, all views of the program—code, value, stack, and graphics—are kept in sync and present a consistent view of program execution (see Figure 3). The Self environment also strives to keep the information presented to the user consistent with the dynamic state of the system.

The principle of immediacy explains why we designed Self and ZStep 95 to have consistent, dynamic views. By providing automatic display updates, the link between the system's state and the display state is removed from the programmer's conscious attention and becomes immediate and unconscious; the programmer unconsciously comes to identify one with the other, reducing the cognitive burden of the debugging task.

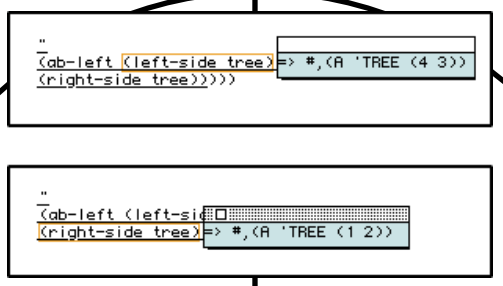

Figure 4. A floating-value window following execution of the code

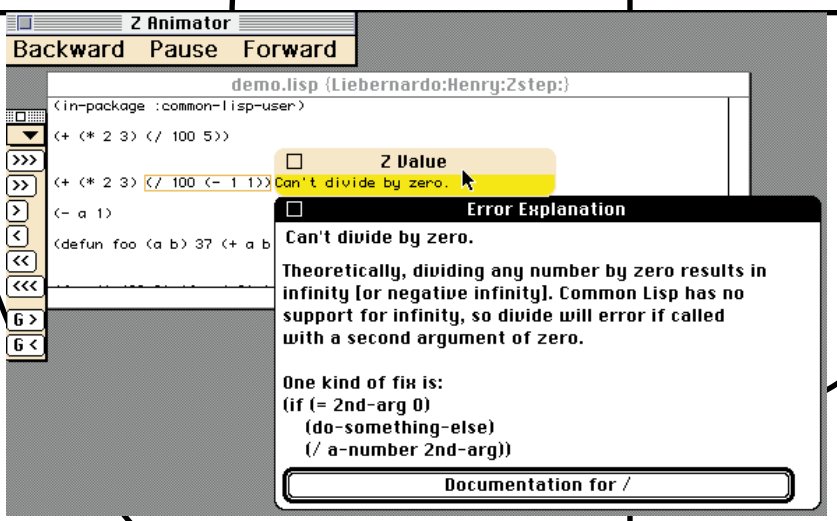

Figure 5. Error messages in ZStep 95

Spatial Immediacy. Spatial immediacy means the physical distance between causally related events is kept to a minimum. The reason is the same as for temporal immediacy: Events widely separated by space on the screen require users to devote more conscious effort to link them, forcing them to shift attention and putting a strain on their short-term memory. Many steppers show the value of the current expression in a fixed window off to the side of the window containing the program code display. This dual-window arrangement causes a "ping-pong" effect as the user's attention bounces from the code to the values and back again. The existence of a separate, constant area that sequentially reflects the values of different expressions adds cognitive distance as well as spatial distance by forcing the user to deal with the changing link between the value display region and the expression being displayed.

ZStep 95 shows a floating window that follows the point of program execution. The code is displayed in an editor buffer, and as the program runs, either forward or backward, fast or slow, the expression currently being evaluated is highlighted and, right next to it, a small floating window shows the code's value, if it has been evaluated (see Figure 4).

The user's attention need never wander to see the value of the current expression. The user need never devote precious short-term memory to the task of remembering which expression's value is being shown. For example, if the user is working backward in the history, the notation "Will be:" along with the value, is displayed, indicating the value lies further along in the history. "Working on it . . ." is shown if the value is only partially computed, so the event is situated in the temporal event stream.

Spatial immediacy is important because it maintains the visual context. Seeing two related pieces of information next to each other fosters making semantic connections in the mind between those two pieces of information. If they are spatially separated, "out of sight" becomes "out of mind." In ZStep 95, a click on the floating window containing the value brings up a data inspector. The code, the resulting

Designers should strive for three kinds of immediacy important for debugging: temporal, spatial, and semantic.

value, and an inspector on that value are all spatially adjacent, reinforcing the feeling of immediacy.

If an error occurs, the error message is shown in the floating window, again spatially located near where the error is manifest, so both cause and effect can be seen at once (see Figure 5). A click on the error message brings up documentation about the error. Another click evokes documentation for the function whose call created the error.

Semantic Immediacy. Semantic immediacy means the conceptual distance between semantically related pieces of information is kept to a minimum. In interactive interfaces, the conceptual distance between two pieces of information is often represented by how many user-interface operations, such as mouse clicks, it takes to get from one to the other. We have already seen some examples of semantic immediacy between code expressions and their values and between error messages and documentation. There are others.

Because ZStep 95 keeps a history of all the graphic events in a program, we can associate each graphic object drawn on the screen with the event that drew it. This association makes it possible to click on a graphic object and go immediately to the place in the execution where the object was drawn (see Figure 6). Not only do we see the source code that drew the object, the entire state of the stepper is backed up to the point in time when the object was drawn, including the appearance of the screen at the time, the values of expressions, and the stack. Similarly, we can point to an expression and jump the stepper immediately to that point in time, keeping all views consistent, including the graphics.

To summarize, the process of debugging a program involves understanding the relationships among the following objects:

- An expression in the source code (not "lines of code")
- The value of that expression at a given point in time
- The graphic output, if any, of that expression
- Expressions executed before or after that expression
- Other expressions or values semantically related, such as the expressions on the stack

ZStep 95's interface is designed to allow programmers to go from any one of these objects to any one of the others with practically no conscious effort. It never takes more than a mouse click. That's semantic immediacy.

Immediacy and Programming Language Design

An environment creates the experience of immediacy by allowing the programmer to make any change at any time and by taking less than half a second to do so, keeping response time below the level of perceivable delay. That way, the program feels as if it is right in the user's hands. We once attended a panel session at Xerox PARC on three

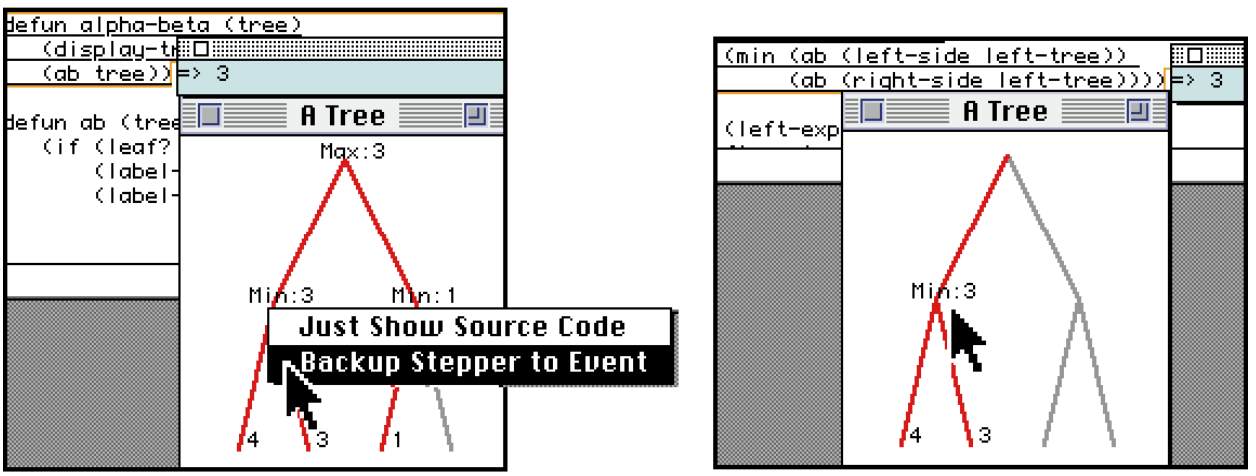

Figure 6. Clicking on a graphic object returns the stepper to the point where the object was drawn

great programming environments: InterLisp, Smalltalk, and Cedar. The Cedar panelist boasted that, since multitasking had been added to Cedar, he now had the luxury of reading his email while waiting for compiles. But Smalltalk users had the greater luxury of having any change take effect in the time it took to press the accept button and see the resulting screen flash. They could then go on and continue debugging without loss of continuity. Just as immediacy applies to both the stimulus and response sides of the user interface, it also applies—at a higher level of abstraction—to both browsing and changing code and data in the programming environment.

Like Smalltalk, Self, APL, and most Lisps let the programmer add a breakpoint to a program and insert extra code to print information, check conditions, or even attempt a bug fix—while the program is suspended. The program can be continued without restarting; the change takes effect immediately, runs at full speed, and can be browsed the same way as any other part of the program. Few other environments—especially those in wide use—boast this level of manipulative immediacy. For example, many C++ systems take minutes to change class definitions in a large program, forcing the programmer to waste mental effort devising workarounds or leading to either boredom (leaving the flow state) or distraction (e.g., reading email). Immediacy at the programming-environment manipulation level is crucial for debugging productivity and underlies many of the positive results reported in the articles in this special section.

Finally, programming-language design is often viewed as high art, mystical religion, or putting together a menu for a restaurant (the principle of orthogonality—the diner can choose sauce independent of meat, giving everyone something they like). One can also appraise the design of a language by returning to the principle of immediacy. In order for a language to bring programs closer to the user, it must be built around a small number of concepts. For example, APL embodies linear algebra with its arrays, functions, and functions-on-functions (aka operators). As a result, to an engineer already familiar with linear algebra, there is hardly any learning curve at all. Just as engineers can write any matrix equation on a piece of paper, they can also write almost any APL expression, unlike many languages that impose extra constraints. For example, APL, though typesafe, has no static type checking. Although the absence of static checking requires the programmer to find and fix some bugs at runtime (the traditional rationale for static typing), its presence compels the programmer to reason much more indirectly about the program. Like a scratch on a skipping CD (or LP), static semantics (e.g., type checking) emphasize the disturbing fact that the program does not execute as written; there is a translation and checking stage that "executes" the program in its own way, so what finally runs is less directly related to what the programmer can write or see.

As another example, macros are a popular variety of static translation on programs, often helping the programmer make up for a weakness in the language design itself. But what happens when there is a bug in the macro? The programmer may wind up staring at "preprocessor output" in order to understand some particularly abstruse and unexpected behavior of the program. In effect, the poor programmer has to debug two programs, the one he or she wrote, and the one that eventually ran. The principle of immediacy dictates that programmers should not have to worry about static invariants unless they want to, but many existing languages force programmers to do so all the time.

Bringing programmers closer to the program, an experience of immediacy helps them understand, change, and ultimately debug. Attaining a reasonable level of immediacy is a precondition for effective debugging and can serve as a useful guide to the design of all aspects of programming environments. C

References

1. Chang, B.-W. Objective reality for self: Concreteness and animation in the Seity user interface. Ph.D. dissertation, Electrical Engineering Dept., Stanford Univ., Stanford, Calif., 1996.
2. Lieberman, H., and Fry, C. Bridging the gap between code and behavior in programming. In *Proceedings of the ACM Conference on Computers and Human Interface (CHI '95)* (Denver, Colo., Apr.). ACM Press, New York, 1995.
3. Lieberman, H., and Fry, C. ZStep 95: A reversible, animated source code stepper. In *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, J. Domingue, M. Brown, and B. Price, Eds. MIT Press, Cambridge, Mass., 1997.

David Ungar (david.ungar@sun.com) is a senior staff scientist at Sun Microsystems Laboratories.
Henry Lieberman (lieber@media.mit.edu) is a research scientist in the Media Laboratory of the Massachusetts Institute of Technology.
Christopher Fry (cfry@shore.net) is the chief technical officer of PowerScout Corp. in Boston.