The Woes of IDEs

Jef Raskin

An epigram: “We may not feel these limitations until they have been lifted from us, just as we often do not know we are sick until we suddenly feel better. Therefore, it is reasonable to expect that future languages will make us feel those limitations of [our present environments] that are not detectable today.”

—Gerald Weinberg

Preaching emanating from the ranks and gurus of the human interface world is slowly convincing management, software designers—and even programmers—that better human-machine interfaces can increase productivity by speeding the work, decreasing learning time, lowering the burden on human memory, and easing users’ physical and mental stress.

This all sounds good, and a lot of it has proved true, but if it’s so good, then why aren’t the human factors people themselves using better interfaces? When they write, most use Microsoft Word (see sidebar). When they program, they use the same integrated development environments (IDEs) as the rest of us. We see lots of effort going into books about how to make Web designs work (though when surfing, I am not sure if anybody is reading all those books), and none at all going into improving IDEs. Such improvements would increase the quality and speed of programming, clearly a desirable end.

So what’s so bad about IDEs? They’ve been problematic for a long time. If you think that it is only recently that human factors in this regard have been given attention, let me quote again from Gerald Weinberg’s book, *The Psychology of Computer Programming*, published in 1971. Its stated aim was “to trigger the beginning of a new field of study: computer programming as a human activity.” He thought that “great strides are possible in the design of our hardware and software, too, if we can adopt the psychological viewpoint.”

He was right.

Weinberg’s book is still worth reading: One of his observations is that “unlike novels, the best way to read [programs] is not always from beginning to end.” Reading most programs feels like tackling a puzzle: You have to worry your way in, backtrack, rethink, and (if you are working from paper) make marginal notes. Donald Knuth, in a 1984 article, later published in a book

Programming speed

AND QUALITY ARE

HINDERED BY POOR

INTEGRATED DEVELOPMENT

ENVIRONMENT INTERFACES.

titled *Literate Programming* (1992), found a way to make programs read properly from beginning to end. Knuth’s innovation

was to write the program in an order that makes sense to humans, and then have a more or less automated process put it into whatever sequence the compiler or interpreter needs to make the program run. He also advocated and provided mechanisms that make documentation simpler.

My own development in the same direction started when I was working on the Apple II in the 1970s. In my method, program segments are embedded in the midst of a word processor document—like raisins in cake—so that the emphasis is on the explanation rather than the code. A simple pre-processor prepares the code for compilation. One such program appears as an example in Susan Lammer’s 1986 book, *Programmers at Work*. The problem has been apparent for at least three decades.

Most current IDEs make adding comments difficult, sometimes painful: You often have to wrap comments by hand, discouraging paragraph-length explanations, or at least discouraging their editing. It is incredible to see antediluvian interfaces in 21st-century products.

The problem of internal documentation is not a minor point. For many programs, paragraphs hardly suffice; essay-length documentation would be appropriate. This is especially true when it comes to maintaining a program. Few are the programmers who can explain their code well enough so that reading it is not incredibly frustrating. A large part of the problem is that only a small part of today’s programs consist of code that is written to implement the specification. Much of what programmers do (as Weinberg was aware) consists of techniques for getting around hardware and software limitations, lengthy declarations, and other formalities required by the system or language design, and other impediments. Programmers rarely note in their comments that a particular technique was chosen to accommodate a requirement of a particular hardware configuration or to get around a compiler oddity.

Some languages are nearly undocumentable (not to

mention that they rarely come with adequate documentation themselves, setting a bad example right from the start). In the past, APL and Forth were given as exemplars of undocumentable languages, but their extreme compactness meant that a single line of code often required a small dissertation. I managed a large project based on Forth, and the result was very readable and maintainable. To achieve this, we had regular code-reading sessions in which one programmer read and commented on each piece of code written by another programmer. In addition, an expert documenter/writer worked alongside the programmers. If he did not understand a piece of code, he would interview the programmers until he could write a cogent explanation.

Incidentally, many bugs and conceptual errors were discovered during this process, which more than paid for itself in decreased debugging time. Some programmers balked at the procedures at first, but all came to be enthusiastic when the project did not slow or founder as completion neared. The project was on time, on budget, and bugless (meaning that the software was released commercially to tens of thousands of users and produced no bug reports).

More modern languages, instead of becoming more maintainable, have gotten less so. This would have surprised Weinberg (see his epigram at the beginning of this piece) and should upset anybody working on or managing a project involving programming today. A prime example of this is Visual Basic (VB). A VB program soon becomes a morass of windows and requires a slog of opening and closing windows to create a program or to follow what is happening in a program. The language is largely unstructured, and writing a program is a wrist-numbing experience. Not only is the environment hellish, but also the language is frustrating to use unless your interface restricts itself to the standard Microsoft widgets. Creativity and imagination are rapidly punished; anything outside the interface norm is either inordinately difficult or impossible to do. The problems with the interface to VB and its reluctance to implement new interface widgets are especially surprising, considering that the person credited with designing VB writes books on interface design.

Some languages, such as Smalltalk and its object-oriented followers, present us with a deluge of classes. It is easier to drown than to swim. Smalltalk itself is simple and elegant. Using it in a practical environment is complex and messy. And, as is usual these days, there is a dearth of documentation. The IDEs for languages such as Smalltalk and Java are conventional GUIs, with their

Withering Word

Microsoft Word annoys nearly every one of its users. Unfortunately, when faced with a repeated stimulus, most of us learn to ignore it no matter how annoying it might be. Most of us have come to accept occasional crashes of our personal computers as a matter of course. This makes any familiar software package, no matter how dismal, seem acceptable.

Word is one such product. It has a bloated set of commands that would take a life's work to master in its entirety. It requires far more mouse clicks and keyboard strokes than necessity demands. It uses memory as if it were free—an 8KB file in the word processor I used to write this column becomes 50KB when cut-and-pasted into Word. Many egregious and difficult-to-undo side effects arise from small keyboarding errors. Finding a system parameter we want to adjust often degenerates to a scavenger hunt among dozens of gray-tabbed dialog boxes hidden behind menus whose names give no hint of which one will contain our prize.

We rapidly tire of having a paper-clip cartoon that pops up when you least want to be annoyed. Word is an example of how our graphic interfaces have gone astray. Yet even most interface gurus use it.

inhumane over-reliance on the mouse. None of this is necessary. It is merely customary.

Open-source development—in many ways a blessing, in that you can dig into the guts of a system if you need to—is also cursed by the oft-undisciplined masses that contribute to most open-source projects. There is little quality control and few who choose to document their work carefully and professionally. Almost no attention is paid to the user interface of the IDE because the programmers who participate generally have not studied cognitive science and are unaware of the difficulties their own IDE designs are causing them.

The computer's truest benefits derive from one thing most of its users never do: programming. Although never will more than a small percentage of users learn programming, that percentage drops to nearly zero given today's opaque programming languages and impenetrable IDEs.

Another impediment to learning to program is the startling lack of manuals. There are extensive online references, and even some tutorials, but it is hard to go back and forth between an online guide and the IDE you are trying to use. Furthermore, almost all tutorials assume that you already know the writer's previous programming

language, and new concepts are explained in terms of that language. It's sayonara, baby, if you don't share the writer's knowledge base. And just try to fix a system when the IDE itself is having problems and you can't access the IDE's help system. There are good reasons for having paper manuals, spiral-bound so they lay flat.

When I work with students, I can see how much time is wasted in learning the ins and outs of complex IDEs and the peculiarities of particular operating systems instead of more generalized programming skills. There is no question that systems have become more complex over time, and that this complexity has to be reflected at least to some extent in our IDEs and operating systems. This is no excuse for making simple tasks complex as a side effect. My son, then in grade school, preferred to learn to program on our old Apple II rather than on a spiffy new Macintosh. Why? Because he just had to type "command-B" and he was in BASIC. Then he could type

```
PRINT "HELLO"  
tap return and see  
HELLO
```

on the display. He didn't have to write a preamble, open a window and specify its size, nor do any other preliminaries. He could get a drawing up in four or five lines of code. The IDE was nearly transparent. A simple problem had a simple solution. He is now a competent programmer in a half-dozen languages and he copes with IDEs of inordinate messiness. Many fewer people starting today will persist in climbing up the steep learning curve needed to get to the plateau where they learn to make an algorithm behave.

The design of IDEs is a problem begging for repair. ☐

JEF RASKIN is best known for his book, *The Humane Interface* (Addison-Wesley, 2000) and for having created the Macintosh project at Apple. He holds many interface patents, consults for companies small and large around the world, and is often called upon as a speaker at conferences, seminars, and universities. His current project, *The Humane Environment* (www.jefraskin.com), is attracting interest both in the computer science and business worlds.

The image shows the cover of the ACM Queue magazine, June issue. The left side has an orange background with the 'acm queue' logo in white, where 'acm' is small and 'queue' is large. Below the logo, it says 'tomorrow's computing today'. The right side has a dark background with a bright, diagonal light streak. The title 'Coming in June Storage Survival Guide' is written in large white text. Below the title is a bulleted list of topics.

acm queue
tomorrow's computing today

Coming in June Storage Survival Guide

- How disks really work, and why it matters
- David Patterson interviews storage giant Jim Gray
- DAFS: A high-performance networked file system