Evaluating Software: What Thoreau Said to the Designer

Paul Taylor

Successful software development is often the result of collaboration among a variety of individuals with different skills and interests. For the past seven years, I have worked with such a group to produce a number of computer programs for use in college-level English classrooms. As a member of this group, I have always considered my primary contribution to be software design and programming. Although I am an active teacher and researcher in the field of rhetoric and computers, my direct technical experience tends to slant my thinking about educational software. I will rely on my colleagues to consider in greater detail specific pedagogical and rhetorical guidelines for evaluating educational software; the following suggestions focus primarily on interface and usability. In fact, some of these guidelines may appear to have relatively little to do with academics, but educational programs cannot be effective while ignoring the general principles of software design. Good educational software must first be good software.

Theoretical Focus

The most important criterion for evaluating software is its theoretical focus. Like a book or an essay, a computer program should constitute a coherent text within a consistent theoretical framework.

A single program cannot do everything for everyone, but it can define a specific purpose and provide tools for achieving that purpose. Naturally, some educators will use software in ways never intended by the authors, just as they modify rhetorical and pedagogical theories to the particular constraints of their own teaching situations. But a coherent theoretical statement—in software as well as prose—at least provides a clear point of departure.

The theoretical focus of a computer program inheres in both its form and its content. All programs have some kind of content—minimally instructions and explanations, if not also more extensive textual, aural, and visual resources. However, this content alone does not constitute the program's theoretical stance. Each program manages the users' actions by establishing possible and recommended operations. The range of permissible actions should be consistent with the program's theoretical orientation. For example, software implementing a collaborative pedagogy should avoid features that give instructors special powers and thereby undermine attempts to foster a student-centered learning environment.

Action

A computer program should allow and encourage users to take actions. A program that simply displays information[1] (even if the information appears in an elaborate multi-media environment) offers no clear advantage over a library. This is not to say that programs cannot or should not be rich in content; on the contrary, on-line resources can significantly enhance learning. It is important, though, that a program provide powerful and flexible ways for users to interact with the information. Search mechanisms (including artificially intelligent agents) make it possible to bypass the software's organization and access information in ways relevant to the user's particular needs. Various authoring tools allow an individual to reshape the program's information into alternate structures. Cut-and-paste commands enable a user to transfer information out of the program and into the user's larger working environment.

These are only a few minimal features important for making an informational database genuinely useful; many other kinds of programs support a much broader range of actions. But any program can easily fall into the trap of passive presentation at some point. Particularly problematic are software designs that force the user to follow a prescribed sequence. Such sequences range from the merely aggravating (title screens with sound or animation that cannot be interrupted) to the genuinely misguided (tutorials that lock the user into a linear

series of screens). Not only do these passive presentations make relatively poor use of the computer's capabilities but they also imply that the user is not very bright and has nothing better to do. Educational software should be a tool that empowers users to accomplish tasks external to the particular computer program.

Management of Complexity

Computer systems are capable of making accessible a tremendous amount of information as well as many different options for working with the information. It is easy for users to become lost in megabytes of textual information and dozens of program commands. Of course, post-industrial society in general has not found a solution to the problem of information overload, and there is no right way to deal with complexity in the specific medium of computer software. But good software provides some means to make sense of the complexity. Perhaps most important is consistency in the interface, a point developed in greater detail below. Other approaches include hierarchies, modifiable levels of expertise, intelligent query mechanisms, and hypertextual structures.

Hierarchical representations can provide a traditional and comfortable means of simplifying complexity. Most program menus arrange commands hierarchically; some programs also provide hierarchical structures for content, such as an outliner that expands or collapses to show a topic at different levels of detail. A few programs offer menu systems that similarly expand or contract according to the user's needs; beginners can use simple menus that show only the most essential operations, while experienced users can select advanced menus that provide more sophisticated features.

Nonhierarchical representations also may help users to manage complexity. Intelligent searches through textual information offer not just a simple string match ("find all the occurrences of the word *tax*"), but a more sophisticated analysis of content ("find all the remarks made by the President's staff regarding the Congressional tax bill"). Such queries enable a user to bring together information that might not be correlated within a hierarchical organization. Another nonhierarchical solution is the use of hypertext, which provides multilinear links between different texts and images. Hypertext makes explicit the multiplicity of relationships attached to any individual word, phrase, sound, or image; in so doing, it provides a guide to at least some of the connections interwoven within a body of information.

Different programs obviously work with different kinds of information; some (word-processing and conferencing software, for

example) depend almost exclusively on text produced by the users, whereas others (on-line handbooks and tutorials) incorporate extensive predetermined content. Some kinds of software require an elaborate command structure, whereas others offer only a few basic options. But all programs need to provide tools for managing complexity—especially software designed for students, whose knowledge and abilities can be expected to change as they use the software.

Consistency

The usability of a program depends significantly on the consistency of its interface. At the very least, a program can be internally consistent. Specialized function keys should perform the same actions throughout the program; one key should not perform two unrelated actions at different times in the same program. Colors, fonts, icons, and locations can be employed consistently to help the user understand different screens easily. For example, whenever the user is expected to write text, the program might provide a blue editing field located at the bottom of the screen or active window; instructions could always appear in white at the top of the screen.

In addition to internal consistency, it is important that a program conform to the external standards established by other software. Of course, standards for software are still evolving because the field is relatively new, and there would be no improvements without deviations from current practices. Nevertheless, a tremendous amount of research has gone into designing effective user interfaces; software that ignores these standards not only rejects the cumulative experience of successful software designers, but also places an extra burden on the users to learn an unfamiliar command structure. One well-known example of the role of standards is the QWERTY keyboard. The letters on modern keyboards originally were arranged to slow down typists in order to avoid jamming mechanical typewriters; alternate keyboard layouts have been proven to allow much faster typing now that jammed keys are largely irrelevant—but the greater efficiency comes at the cost of retraining everyone who learned to type on the QWERTY keyboard. Although the current keyboard layout is antiquated, it serves as a standard that enables typists to move easily from one system to another.

Similarly, software standards enable users to learn new programs quickly and integrate them into their other activities. Several features are likely to remain central to software produced in the next decade, including pull-down menus, dialog boxes, icons, and moveable, resizable windows. Often these elements are described collectively as a Graphical User Interface (GUI) because they were pioneered in a graphics

environment, which can provide attractive visual details such as three-dimensional shadowing underneath buttons. It is important to note, however, that the essential functionality can be realized without fine-grained graphical decoration and even without a mouse, which is frequently assumed to be an integral part of a GUI. All programs can implement the basic features of a menu-driven interface.

In addition to screen designs, alternative input and output devices are increasingly important. The mouse is now standard on most new microcomputers, and other devices are becoming more widely implemented. Software certainly should provide mouse support, and forward-looking designs also will respond to touch screens, light pens, and voice commands. Of course, it is difficult to predict which new technologies will flourish—ten years ago it might have seemed like a good idea for all software to support joysticks. Nevertheless, these alternative technologies open up the possibility that all users may find new, more efficient ways to interact with computers. In addition, they make computers more accessible to people with disabilities. As these devices become standardized, software will be expected to integrate them into the interface.

Many other minor conventions establish uniformity within a program: The ESC key cancels a request or backs up a step, F1 provides online help, "File" and "Edit" options are grouped together in a prominent position on the menu. Such details are too numerous to list completely; furthermore, they may change as new techniques are developed. In the end, recognizing whether a program effectively observes conventions requires the evaluator to become familiar with a broad range of commercial and educational software.

Connectivity

With the possible exception of games, most software is not an end in itself. Computer programs should be considered not as self-contained systems, but specialized components within a larger suite of computer-based activities. Each program needs to be able to transfer information in and out of its own workspace; specifically, the program should allow the user to connect to other programs, other information databases, and other users. The simplest level of communication between programs is a text-only transfer; for example, a student might use a heuristic program to explore a topic, save the work as a text-only file, and then import the prewriting into a word-processing program for possible incorporation into an essay. Although this simple level of interprogram communication is essential, it is not enough in an era of computer-based documents that may easily include various type styles,

graphics, sounds, and video clips. Programs can facilitate more complex transfers by observing appropriate standards for storing nontext information. In addition, individual programs can tie into resources provided by the computer's operating system. Although the specific technology will change, two current examples are the "Publish-and-Subscribe" feature of the Macintosh System 7 operating system and the "Dynamic Data Exchange" protocol available in Microsoft WINDOWS 3.0; both of these features enable different programs to share complex information—as long as the programs have been written to take advantage of the resource.

Software also can enhance information exchanges by recognizing and utilizing network services. As more computers become attached to local and wide area networks, it becomes increasingly important for programs to avoid technical conflicts with network software and to provide various "hooks" to network capabilities. Networks can give access to diverse information databases and also link users who may collaborate formally or casually. Programs that recognize these potential connections can smooth the process of gathering and distributing information and cooperating on shared tasks.

Feedback

Good software lets the user know what is going on. When the user presses a key, clicks a mouse button, or provides some other kind of input, the program should respond immediately. Any action that cannot update the display immediately should provide an intermediate indication that something is happening—at least through a simple message or cursor change. Longer processes should provide some kind of indicator to show the progress on the action—for instance, a graph or number showing the percentage of the task completed.

Other displays can help the user keep track of current operations and opportunities. For example, communications software can list the names of everyone currently active in an electronic conference. Information systems can keep track of previous search terms and provide visual maps to help orient the user within a web of textual connections. Every program can list existing files when saving or retrieving—preferably distinguishing between files created by the program and all other files. Of course, the trick is to make all this information easily available without cluttering the screen and overwhelming the user.

Flexibility

Educational software should be flexible from the perspective of both the instructor and the student. For the instructor, flexibility means that the program's content and default appearance can be changed. Tutorials, drills, heuristics, and even on-line reference works all need to provide simple, reliable mechanisms for the teacher to modify existing content and insert new content. New content should not simply replace the original text (unless the instructor chooses to do so); the software should be designed to incorporate additional material clearly and seamlessly. Ideally, all text that appears in the program should be modifiable so the teacher can customize it for different languages and knowledge levels.

For individual students, software should provide the flexibility to customize the display and control the sequence of events. Each user should be able to establish personal preferences regarding screen colors, fonts, sound levels, printers, locations of personal files, and levels of expertise. In a network environment, where many individuals may run a single copy of a program, the software should ensure that one individual's preferences can be stored without losing the preferences of other users. Although it is possible for students to waste time playing with these options, the freedom to personalize the program can help them to create a more productive environment. For visually impaired users, the ability to change colors and fonts—for all text that appears on the screen, not just text produced by the user—can make the difference between a valuable application and a waste of time.

The users' control of the environment also should extend to the sequence of events in a program. With a few exceptions (such as some specialized testing procedures), software should generally give the user the ability to choose where to go next, including a simple return to the previous step. If a program presents information in a timed sequence, the user should be able to modify the speed of the presentation. It should be easy to stop at any point in a program, save work in progress, and resume later at precisely the same point where the user previously quit. Flexibility and user control properly place the primary focus on the person rather than the software.

As students sit down to face a new computer program, they may wistfully recall Thoreau's exhortation to "simplicity, simplicity, simplicity!" Feature-laden programs enable users to take advantage of a wide range of resources, including network connectivity, multimedia information, and tools for restructuring information—but such power sometimes comes at the price of a high learning curve. In order to be

usable in the classroom, software must manage the complexity it creates; it should be internally and externally consistent, and it should be flexible enough to provide individual users different ways of simplifying the environment. Even more important than usability, however, is the way in which a program is integrated into the curriculum. Although an up-to-date interface is vital to any application, it does not automatically make good software any more than a book printed on acid-free recycled paper makes good scholarship. Theoretical focus remains the primary concern; in evaluating software, instructors must understand what the program is designed to do and determine how that purpose fits into the larger goals for the students in the classroom.

Note

1. I use the term *information to* refer to text, sound, and graphics that may be accessible through a computer program. That term seems rather dry, as if it referred to technical specifications for a computer chip or an airplane's landing gear. Actually, I am borrowing the word from information theory, pioneered by Claude Shannon in the 1940s. Early work in information theory explicitly separated information from meaning; Shannon and others were not addressing epistemology, but the transmission of data. However, the field has grown to consider how meaning arises and changes through the relationship between a message (the text/sound/data sent by an author) and the accompanying "noise" (the unexpected, the ambiguous, the context). Both the message and the noise constitute information, and a mixture of the two provides the greatest information. Recently, research in chaos theory has suggested that noise in a message can cause the system to self-organize at a higher level; thus the disorder and uncertainty are not extraneous distractions, but integral to the creation of meaning. It is this notion of complexity in symbolic communication that leads me to use the term *information*.

Paul Taylor is an Assistant Professor of English at Texas A&M University in College Station, Texas.