Design Models for Computer-Human Interfaces

Donald R. Gentner
Sun Microsystems

Jonathan Grudin
University of California, Irvine

Human interfaces are a Rorschach test for designers—their inner views and biases are unconsciously reflected in the types of user interfaces they construct.

Computer-human interface design has been recognized as a distinct field for only a little more than a decade, but the design of interfaces to control mechanical devices has a much longer history. The interface-design models used in these mechanical systems play similar roles in computer systems, despite the obvious differences between the two types of systems.

There are many ways to control a given mechanism. Whether consciously or unconsciously, every interface designer chooses a model that forms the basis for how the mechanism is controlled. Two principal approaches are the *engineering model* and the *user-task model.*

Suppose you want to adjust the flow and temperature of water emerging from a faucet. From an engineering standpoint, the problem is to adjust the flow of hot water and cold water and then combine the streams to produce a mixture with the desired temperature and flow rate. This engineering model has been mapped directly to the user interface of the faucet in Figure 1a—one knob is attached to the hot water valve; the other to the cold water valve.

In contrast, the interface of the faucet in Figure 1b is based on a user-task model—moving the handle up or down controls the flow and moving the handle left or right controls the temperature. Clever mechanical design converts these movements into valve operation and lets the user focus on the task.

There is no "best" way to design a user interface, however. Interface designers must be aware that a user interface can be based on any of several models, that each model has its advantages, and that their job is to choose the approach most suitable for the project at hand. In this article, we examine the models underlying computer-human interface designs by considering a wide variety of systems, including many from areas outside of computing. These noncomputer examples can be instructive because they are simpler and thus clearer. They also provide some helpful detachment and perspective for those of us who are immersed in computers.

CONTRASTING APPROACHES

Radios provide a simple example of the contrast between interfaces based on engineering models and user-task models. Radio stations broadcast at different frequencies. Every radio receives all these signals, but also has a resonant circuit that is adjusted to a particular frequency. Many radio interfaces are concretely based on this engineering model. To select a station, the user simply turns a knob attached to the variable capacitor in the resonant circuit. The knob is calibrated with numbers that correspond to the resonant frequency of the tuned circuit (such as 550 kHz).

Other radios have a push-button interface. Once a push-button radio is programmed, users can select a station by pushing the corresponding button. The user's intention to listen to a particular station maps directly into the press of the appropriate button. This interface better reflects the user's model of the task, but it provides access to only selected stations.

The first electronic hand-held calculator, the Hewlett-Packard HP-35,

0018-9162/96/$5.00 © 1996 IEEE

also had an interface that directly mirrored the underlying engineering model, in this case a stack architecture. To add 27 and 56, for example, the user pressed "27 ENTER" (pushing 27 onto the stack), then "56 ENTER" (pushing 56 onto the stack), and then "+" (adding the top two numbers on the stack). Calculators that use this reverse Polish notation still have a loyal following, but most calculators today use algebraic notation, so users can enter the more conventional sequence, "27 + 56 =," even though the calculator may still implement the operations with a stack architecture.

Figure 2a shows an example of a programmable remote control for home audio-video equipment that must have seemed perfect to its designers, a group of brilliant computer hardware engineers who were designing their ideal controller. The Core remote control could be programmed to store up to 16 "pages," each containing 16 command sequences. The hexadecimal nature of the underlying algorithm shows through in the keys labeled 0 to F. Even the typeface used to label the keys is based on a 7-segment LED display. By contrast, the programmable remote control shown in Figure 2b has keys labeled with typical user-task terms, such as VOLUME, CHANNEL, and STOP.

Engineering model

As you might expect, basing an interface on the engineering model has the greatest benefit for engineers and developers. After all, if your task is engineering, the task model is essentially the engineering model! Solutions based on the engineering model can be simple and elegant.

In addition, just as the tuning dial lets you access all radio stations, being able to access a system's full functionality helps developers debug and fine-tune it. A system used primarily by engineers and hobbyists might also need to provide this complete level of control.

Systems with interfaces that reflect the underlying architecture can be easier for the engineer or knowledgeable user to troubleshoot, maintain, adapt, and enhance. Such interfaces are especially important if the system is used in a wide variety of environments. If most users will eventually need to access the full functionality of the system, it may be better to begin introducing them to the engineering model at the outset. And if the system malfunctions and the interface has provided users with a good model of the underlying mechanism, problems will be easier to diagnose, repair, or work around.

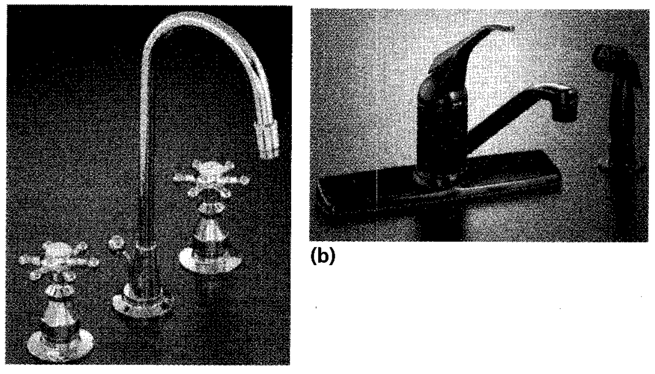


Figure 1. Two designs for kitchen faucets. (a) The user interface for this faucet is based on an engineering model—the faucet handles directly control the hot and cold water flows. (b) The interface for this faucet is based on a user-task model—moving the handle up and down controls the combined flow rate; moving it from side to side controls the temperature.

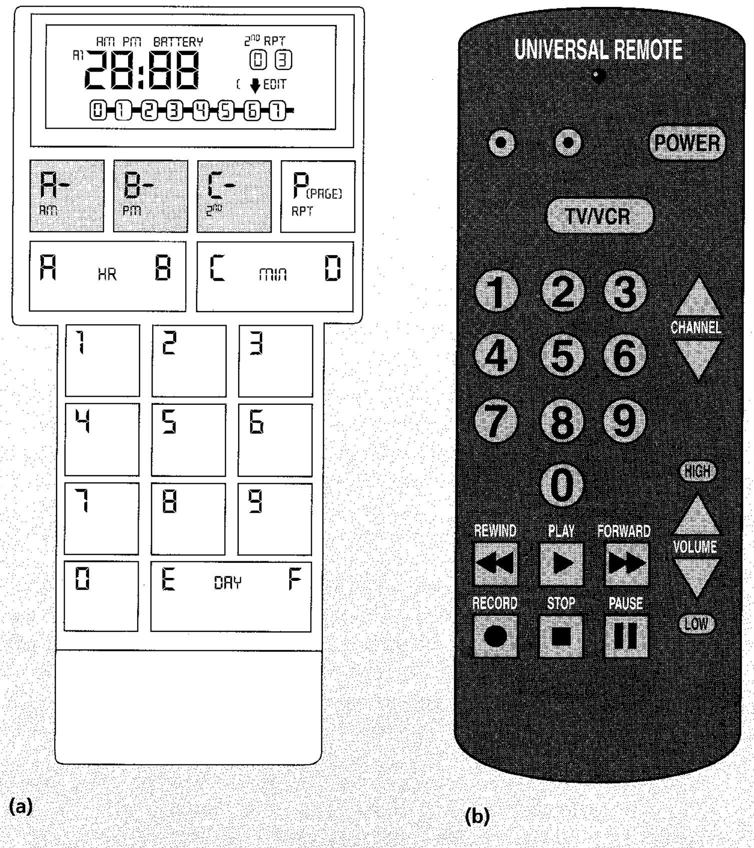


Figure 2. (a) A remote control device from Cloud 9 directly reflects the engineering model, with buttons labeled with the hexidecimal code 0-F. (b) The more familiar remote-control interface is labeled with user tasks (rewind, play, TV/VCR).

The Apollo astronauts demanded and received more access to the on-board computers and software that controlled their flights. This access proved crucial in cases of system malfunction.

User-task model

Basing an interface on the users' task provides a complementary set of user benefits. Users already have a model of the task, so an interface based on the engineering model forces the user to translate between two models. A successful task-based interface would let them work with a single model.

The primary benefit of the user-task model is faster, more effective learning. Operations that users must perform should be directly accessible in the interface, as with a push-button radio. An interface built with a sense of the task should also enable very efficient use of a system by experienced users, through appropriate forms of navigation, optimal arrangement of features, provision of useful shortcuts, effective help on advanced features, and so forth.

A task-based interface can guard against errors that are particularly likely or costly. For example, it can prevent users from overwriting important parts of the memory. Similarly, an interface based on the task model can promote good policy, as in text processors that remove multiple spaces between words or sentences. The drawback is that supporting an additional user-task model adds to the cost and complexity of development.

Selecting an approach

Both approaches have strong advocates. Some argue that users should be introduced to the engineering model from the beginning because they will inevitably have to deal with the underlying system. Others argue that the optimal interface is always one that directly corresponds to a user-task model. Neither extreme position seems viable today.

Higher level models can be very successful. Graphical interfaces are not based on engineering models and generally strive to hide the underlying mechanism. Many users of desktop interfaces, like Windows, that run on traditional operating systems, like MS-DOS, find no reason to leave the desktop. The growing number of users who are basically unaware of or indifferent to system mechanisms is fueling a movement toward interfaces based on user-task models.

But neither is it always correct to base the interface on user-task models. Users' tasks have often been shaped by the tools they have used in the past, and blindly adapting an interface to an existing task can lock users into obsolete behavior. Even when tasks are nominally the same, differences among people and work situations mean that one user-task model will not fit everyone. Thus, modeling the interface closely on a single user task can restrict the range of settings for which the application is well-suited. A user might be unable to perform a task that the designer had not anticipated, even though the machine is fully capable of performing the new task. For example, we once used a convenient viewer for reading Usenet messages that provided no way to go back and reread an old message, even though the message was still on the news server.

Good interface design requires a balance between conforming to the machine mechanism and conforming to the users' conception of the task. Slavishly imitating the way the task was performed in the past can be a bad idea because that performance was shaped by an earlier generation of tools. Over the long term, a new technology must find the proper balance between forcing users to accommodate in order to exploit the strengths of the new technology and adapting the technology to the users' preexisting model of a task. This challenge is particularly relevant to computer interface designers. A computer's adaptability places relatively few restrictions on the interface, and it is thus relatively easy to simulate the previous technology.

HISTORICAL EVOLUTION

The engineering and user-task models are not fixed. Technical innovation produces major changes in the system mechanism through expanded hardware capabilities, novel input and output modes, and new software. At the same time, users' work environments and tasks evolve to fit the available tools. The evolution of the interface tracks a compromise between the changing engineering and task models, producing a loosely coupled coevolution among the mechanism, interface, and task, driven by both technological and environmental change.

Early interfaces

The first interfaces for a radically new technology tend to be closely tied to the engineering model because that is the easiest to implement and because early users are usually technically oriented anyway. The first computers were programmed by plugging wires into large patch panels to directly alter the electrical circuits. Stored-program computers were initially programmed in machine language. Only later, with the introduction of high-level languages, were programmers able to work with more familiar mathematical and logical constructs. Although high-level programming languages and task-based applications had been developed in the meantime for mainframes, history repeated itself when the dedicated users of the first Altair home computers used toggle switches to enter machine-language programs. Only later did microcomputers offer higher level languages, such as Basic and C, and finally task-oriented, end-user applications.

Programming languages went through a similar evolution. Assembly languages have statements that correspond closely to the computer's machine language. For example, the following sequence of instructions would add 17 to a variable named `total` and store the answer in a variable called `bigtotal`.

```
lda total     (load total into
                accumulator a)
adda 17.0     (add 17 to the value in
                accumulator a)
stoa bigtotal (store accumulator a in
                bigtotal)
```

Each assembly language statement mirrors a CPU operation. The same operations written in a high-level language like Fortran or C, however, would be written in a form closer to the way a user might think:

```
bigtotal = total + 17;
```

Simulating previous technology

A few years after a new technology is introduced, manufacturers often try to broaden the market by building interfaces that simulate a previous technology. The goal is to reduce or eliminate learning, making the technology widely available to people who are unwilling to adapt to a new system. Users are often reluctant to invest in learning a new system if they already know ways to perform a task using old technology. Wendy Mackay[1] found that an extremely common use of customization in computer applications was to mimic features in the previous version of the software.

One of the most delightful examples of an interface based on the preceding technology is the Phelps tractor, introduced to farmers in 1901 as a replacement for the horse. The Phelps tractor could be hitched to a carriage or wagon, and farmers used a pair of reins to control the tractor just as they would control a horse, as Figure 3 shows. The tractor was steered by pulling on the appropriate rein. When both reins were loosened, it went forward; when they were pulled back, it slowed down and stopped; and when they were pulled back harder, it backed up.[2] Automobiles controlled by steering wheels and levers had been commercially available in the United States for about 10 years before Phelps tried to emulate the horse interface. Presumably the designers of the Phelps tractor believed that farmers would find it easier to control than contemporary motor vehicles, with their unfamiliar levers and knobs.

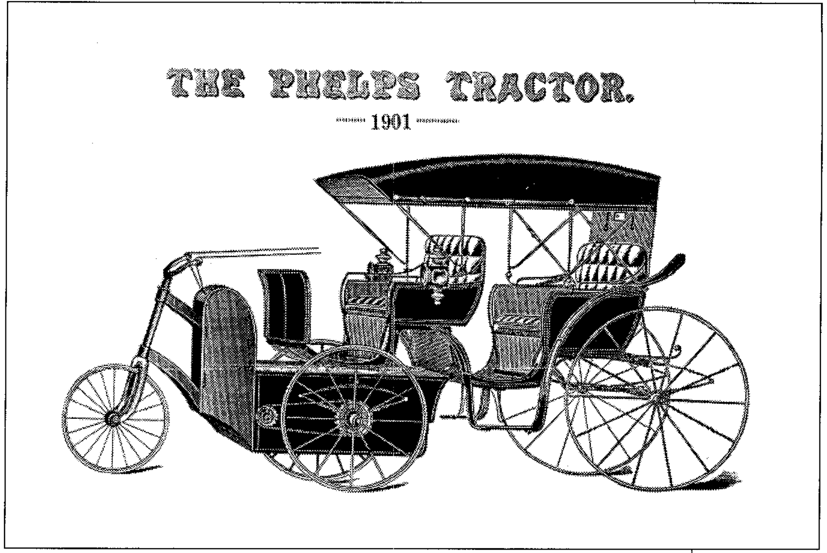


Figure 3. Phelps tractor. The user interface is based on the previous technology—the horse.

Farm tractors are not the only example of this approach. Although computers are sometimes heralded as ushering in the paperless office, most PC interfaces are now based on a traditional office metaphor, complete with documents, folders, filing cabinets, and wastebaskets. In recent years, attempts to imitate the old office have been taken to greater extremes, as in Wang's Freestyle interface[3] and the Magic Cap desktop, shown in Figure 4.

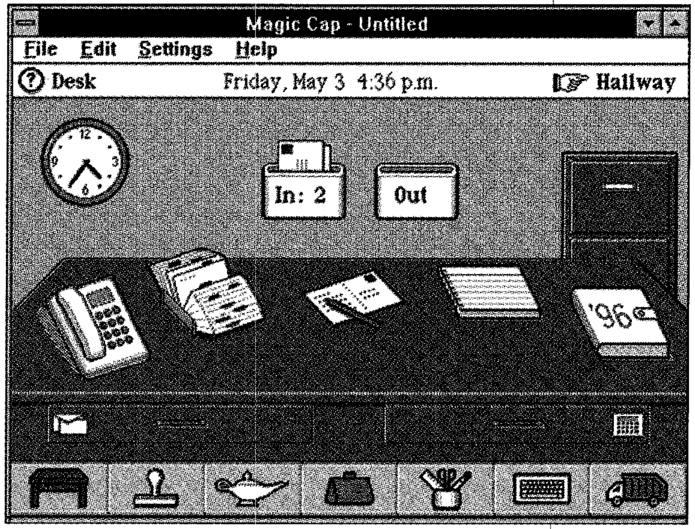


Figure 4. General Magic's Magic Cap interface is based on the familiar model of an office.

A similar example in computer software is Fractal Design's Painter, a very successful program that attempts to faithfully simulate a wide range of paintbrushes, pencils, pens, papers, watercolors, and other tools and materials of the traditional fine art painter.

Evolving interfaces

We already see examples of this general historical progress in human-computer interaction. Consider one of the first computer applications to succeed widely outside engineering settings: word processing. The first command-driven text editors were based closely on the underlying system model and the need to write programs line by line. Later word processor interfaces were designed to support user tasks by carefully reflecting an existing technology, electric typewriters. In 1983, three IBM researchers recommended: "...we should try to design in more analogies between the [text] editor and the typewriter."[4] This strategy succeeded in expanding the user population, but desktop publishing systems have carried the interface and capabilities far beyond the typewriter. Today, hypertext interfaces exploit computer technology in radically new ways. Writing itself is evolving.

WORKING WITH LAYERS

In all machines that have manipulatable controls, part of the device is devoted to connecting the controls to the underlying mechanism. Even a conceptually simple interface based on the engineering model requires connections to control points. In early cars, the carburetor throttle and choke valves were attached to dashboard levers, a direct mapping that required only simple engineering. In general, connections to control points are necessary during development to aid in fine-tuning or debugging, and these connections can be cleaned up in the final user interface.

The "How much should users see" sidebar addresses a recurring question in designing multilevel systems: How much of the underlying mechanism should be revealed to users?

Basing an interface on a task model complicates engineers' lives in three ways: First, they must define the task model. One way to do this is to perform a task analysis, which often involves user consultation and iterative prototype designs.[5,6] This can be a major undertaking in itself. Here we focus on the other two challenges: Understanding the engineering and task models and linking them.

Understanding models

Engineering complex appliances such as car radios is often done through a division of labor. A materials engineer knows the properties of the materials used to design the capacitors, resistors, and other components. A radio designer knows the properties of these components but not of the underlying materials. An automobile designer knows the properties and requirements of the dashboard radio that will connect to the electrical system, but could know nothing about radio internals.

Computers also have several levels. The deepest level is the hardware, including the processor. Few programmers or software engineers work at this level, which is controlled by programs written in machine language. The next level is controlled by the more powerful commands of assembly language, which are automatically translated into machine language. Next, high-level languages are built on an assembly language. And toolkits and libraries of routines are on yet a higher level.

A different engineering model of the system exists at each level. Clearly, the engineering model at the level at which most interface developers work will most likely influence the interface.

The effects of these different levels often seep through to users. When programmers worked in assembly language, for example, error messages sometimes specified register numbers or hexadecimal memory addresses—information useful to engineers but perplexing to most users. As interface developers shifted to higher level languages, such error messages were sometimes replaced with messages such as "syntax error," "string overflow," or "disk read error." Again, these are more informative to developers than to users. The intrusion of engineering terminology is only one way that the engineering model can affect the interface.

Linking the models

Now consider the link between task model and engineering model in this multilevel context. If we consider the most basic mechanism—the hardware—the entire structure of assemblers, compilers, toolkits, and other software constructs is part of the linkage to the task model, similar to the rods and pulleys in radios. In such a system, the conceptual distance between the hardware and the interface based on the task model is great, but no one engineer has to bridge it. Engineers generally work at one level, but are still aware of the model at the next highest level. A person creating a library of C routines carries out engineering tasks at the level of the C language, but should also be aware of the tasks of the programmers who will be the library's users.

Thus, the complexity of linking the final users' task model and an engineering model depends on a developer's level. When the linkage complexity is very great, it can be reduced by creating a new, higher engineering level. This level can incorporate elements of the user-task models or allow the task model to be represented more easily. Once the effort

How much should users see?

A recurring question in designing multilevel systems is: How much of the underlying mechanism should be revealed to users? Part of the answer is that it depends on the users' tasks. If most users are engineers, programmers, or hobbyists, they might want deeper access, although perhaps not down to the level of machine code.

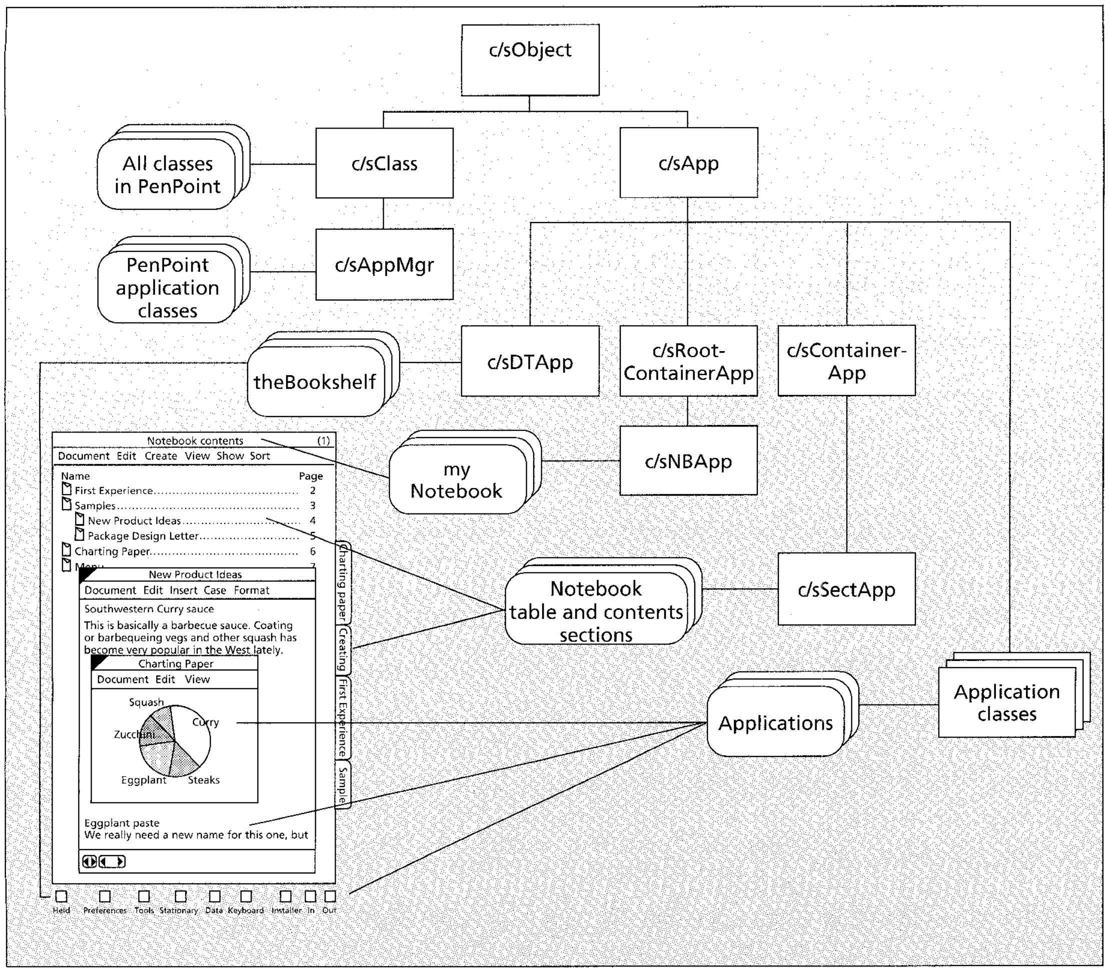
One reason to reveal more of the system's mechanism is to help users learn the system faster. Etienne Wenger[1] discussed users' views of systems in terms of *black-box* versus *glass-box* models. A black-box model hides the system mechanism completely. Users learn that certain actions cause the system to perform in a given manner, but the connection between input and output seems arbitrary. A glass-box model reveals the mechanism completely. Users can reason about the relation between input and output and predict how different actions will affect the system behavior.

The risk in using the glass-box model is that you can overestimate the amount of information users want and provide information that is primarily of interest to other engineers. For example, the Mosaic World Wide Web browser periodically announces the number of "bytes downloaded" when it establishes a connection. This is not particularly useful information to most users, and it conflicts with Mosaic's model of a "window onto the Internet." Because the files are not permanently downloaded, their size is of limited significance once access is complete.

The greater flexibility of computers can reduce the need to rely on underlying mechanisms. Radio users have no intrinsic interest in the frequency at which a station broadcasts. However, the number and limited broadcast range of stations precludes putting names or call letters on a dial, so listeners have to learn a station's frequency in order to find it. With a small amount of memory and processing, developers could now provide an interface that eliminates the need to rely on that engineering detail. The same is true of telephone numbers—an engineering detail that automatic dialers can remove from the telephone interface.

Reference

1. E. Wenger, *Glass-Box Technology: Merging Learning and Doing*, IRL Research Abstract 1, Institute for Research on Learning, Palo Alto, Calif., 1988.



Southwestern Curry sauce

This is basically a barbecue sauce. Coating or barbequeing vegs and other squash has become very popular in the West lately.

Eggplant paste

We really need a new name for this one, but

Figure 5. The class hierarchy of the PenPoint application is reflected in its interface. Each item in the user interface directly corresponds to a program object in the code.

of creating a new language toolkit or environment is over, engineers working at the new level can more easily implement the interface, the link to the users' task model.

This is precisely how levels are built up over time. Engineers and programmers work at each new level, but the process is driven by the need to implement the functionality and interfaces dictated by users' tasks. Each new level reduces the gap between the highest engineering model and the task model those engineers will support.

Reducing the gap might simplify the engineer's task only temporarily. By providing greater efficiency, a new level can encourage new features and functions to support more numerous and complex tasks. These lead to more complex task models, increasing the complexity of the interface-mechanism linkage. Eventually, this complexity could become great enough to justify the effort of creating yet another level. We have seen no reason to suppose that this accumulation of levels will stop.

Changes can also propagate down through engineering levels. New, higher level languages favor the adoption of different assembly languages. One benefit of the shift to RISC architectures is that developers can now better understand program demands.

LOOKING TO THE FUTURE

Gerhard Fischer[7] has noted the historical sequence of higher level engineering in programming, and sees his work on "high functionality design environments" as a continuation of this sequence. The notion is that the increasingly strong functionality and interface demands within a specific application domain make it more efficient to abandon general-purpose languages and environments in favor of specialized environments created for specific domains. A design environment to support network designers and administrators, for example, can include domain-specific information and more sophisticated advice for developers working within that domain. This is an extension of trends already evident in the existence of higher level languages and libraries that favor one or another broad class of applications.

Object-oriented languages are another approach that could reduce the gap between developer and task model. Objects meaningful to users can be directly represented in the language, allowing developers (and perhaps users themselves) to manipulate the representations to match user requirements as they evolve. Figure 5 shows an example of this. The hierarchy of the application directly reflects the class hierarchy.

Mastery of tool versus mastery of task

The shift from engineering to task models reflects the fact that, for most users, the interface is a means to an end. It's like a blind person's cane. The cane handle is the interface to the tool; the tip of the cane is the interface to the world. The handle is important, but once it is mastered, the activity at the tip of the cane is more important. As users master a tool, they draw on prior experience. Later, when they have become more comfortable with the tool, their interest shifts toward mastering the tasks the tool is used for. Interface designers can follow this shift.

Of course, successful applications have many users, not all of whom are in the same place at the same time. A developer can't please everyone. It's great to provide some support for novices, experts, casual users, power users, the enthusiastic, and the reluctant, but it's also expensive. Developers must estimate which users are most important at a given time. It's dangerous to assume that knowledgeable enthusiasts are most important because you are a knowledgeable enthusiast, or that novices are most important because novices participated in a usability test, or that the current interface design is adequate for tomorrow because it is doing well today.

ADVICE FOR DEVELOPERS

Often, the path of least resistance is to model the interface on the mechanism. This is familiar to the developers; it's what they themselves usually want to work with. What's more, they find encouragement in often-repeated guidelines:

- "Build a consistent interface." This can be good advice, but "consistent" is often interpreted to mean consistent with the underlying engineering model, the software architecture.
- "Strive for simplicity." Another good goal, but it is generally simpler and more elegant to base the interface on the underlying mechanism, which is not always a good idea.

We offer some other advice:

- *Place yourself in the historical evolution.* Examine the current state of your interface. Take a good look at those architecture diagrams. How closely does the interface mirror the underlying system? Identify similar technologies. How many of their features are reflected in your interface? Recall the last time you examined workplace practices at sites using your systems. Has your interface evolved since then? Remember, it is not necessarily good or bad to be at any one point in this progression. The exercise is to first figure out where you are, and then ask where you should be.
- *Determine where you should be.* Most systems are not entirely new, but novelty is more widespread in software than in most fields. Furthermore, novelty is in the eye of the beholder. Marketing a familiar kind of product to a completely new audience requires rethinking its interface. Not all novel applications are designed to be used primarily by engineers or hobbyists, but some are. Make a realistic appraisal of the early adopters. If they value control, consider basing your interface on an engineering model. Some developers think their system will be used as-is, but in fact their system will be modified by other programmers. In these cases the original developers should provide tools that enable the programmers to access the underlying system. On the other hand, if the prospective users are unfamiliar with a system they will use to do their work, consider using a task model, keeping in mind that the task will change when a new system is introduced. In these situations, basing the interface on a familiar technology or practice can be the surest route to acceptance.

  If prospective users are either more computer-savvy or have experience with an earlier version or a similar application, it is crucial to understand the work activity. These interfaces must move beyond familiar technologies and systems, even if this requires engineering changes. Often, you might want to design for a range of users, but this complicates the situation. Multiple interfaces, while possible, add complexity for developers and users alike. It is worthwhile to make a strong effort to identify the key user population.
- *Pay attention to trade-offs.* When developing systems you must balance competing pressures. An interface design is influenced by both tasks and constraints, not to mention schedule, budgets, marketing, and numerous other factors.[8]

IN GENERAL, AN INTERFACE BASED ON the engineering model allows full access to the system's capabilities, whereas an interface based on the task model is easier to learn and use but provides access to only a subset of the system capabilities. In weighing the costs and benefits of each approach, you must look beyond the immediate task to consider the broader context of use. For example, applications intended for people who infrequently use a system must focus on ease of learning and remembering. This implies that interface design should be based on the users' existing task model, even if this increases the complexity of the underlying system design or decreases its power. The system mechanism will not be relevant to infrequent users, so it should be hidden as much as possible.

On the other hand, it is reasonable to expect frequent users to invest more time and effort in learning a new system if—and this is a big if—they see major benefits in adapting to the system. In particular, gaining an aesthetic appreciation of the engineering model is not seen as a major benefit by most users. As an example of how frequency of use affects interface design, information kiosks placed in a public shopping mall must rely on very simple user task models, such as "point at what you want," whereas information systems used on a daily basis can provide a powerful database query language.

A good case can be made for including on the design team some individuals—perhaps prospective users—who remain naive about the underlying system architecture. Finally, users' tasks in broad work settings must be continually examined, for they will change in parallel with technological change and provide opportunities for new interface designs. Interface designers will play an important role in the foreseeable future. ❙

Acknowledgments

We thank Jakob Nielsen, Don Norman, and the reviewers for helpful comments on earlier drafts.

References

1. W. Mackay, "Triggers and Barriers to Customizing Software," *Proc. CompuOter-Human Interface '91*, ACM Press, New York, 1991, pp. 153-160.
2. F. Clymer, *Treasury of Early American Automobiles*, McGraw-Hill, New York, 1950.
3. S.R. Levine and S.F. Ehrlich, "The Freestyle System: A Design Perspective," in *Human-Machine Interactive Systems*, A. Klinger, ed., Plenum, New York, 1991, pp. 3-21.
4. R.L. Mack, C.H. Lewis, and J.M. Carroll, "Learning to Use a Word Processor: Problems and Prospects," *ACM Trans. Office Information Systems*, July 1983, pp. 254-271.
5. J.D. Gould, "How to Design Usable Systems," in *Readings in Human-Computer Interaction: Toward the Year 2000*, R. Baecker et al., eds., Morgan Kaufmann, San Mateo, 1995.
6. *Design at Work: Cooperative Design of Computer Systems*, J. Greenbaum and M. Kyng, eds., Lawrence Erlbaum, Englewood Cliffs, N.J., 1991.
7. G. Fischer, "Domain-Oriented Design Environments," *Automated Software Eng.*, June 1994, pp. 177-203.
8. J. Grudin, "Systematic Sources of Suboptimal Interface Design in Large Product Development Organizations," *Human-Computer Interaction*, No. 2, 1991, pp. 147-196.

Donald R. Gentner *is a senior staff engineer at SunSoft, a Sun Microsystems business, where he designs human interfaces for end-user computer applications and explores new directions for computer-human interfaces that embrace the Internet. Previously, he designed human interfaces at Apple Computer and Philips Laboratories, and conducted research in cognitive psychology at the University of California, San Diego. He received a BS in chemistry from Rensselaer Polytechnic Institute and a PhD in physical chemistry from the University of California, Berkeley.*

Contact Gentner at SunSoft, 2550 Garcia Ave., MTV21-225, Mountain View, CA 94043-1100; don.gentner@sun.com.

Jonathan Grudin *is an associate professor of information and computer science at the University of California, Irvine. He previously taught at Aarhus University in Denmark, developed software at Wang Laboratories, and carried out research at MCC. He is in the Computers, Organizations, Policy and Society group at Irvine, where his interests include computer-supported cooperative work and human-computer interaction. He received a BA in mathematics-physics from Reed College, an MS in mathematics from Purdue University, and a PhD in cognitive psychology from University of California, San Diego.*

Contact Grudin at Information and Computer Science Department, University of California, Irvine, Irvine, CA 92717-3425; grudin@ics.uci.edu.