him, he will be proceeding into the jungle on false pretenses, so we cannot do that. If we edit his work and assure that he knows about it, we will have had to use an error message or a confirmation dialog box. This is also not acceptable. The only choice we have is to run along behind our brave user, making sure that he doesn't come to harm. We keep track of his path into the jungle; we remember each of his actions; we assure that each action can be cleanly reversed; we assure that no collateral information is lost. Essentially, we maintain a clear audit trail of his actions. This is why I formed the axiom: Audit, don't edit.

Audit, don't edit

If you can't bound input data with gizmos, just go ahead and accept whatever the user gives you. Then keep track of what you did and didn't get, and if anybody demands that things get straightened out, you will have full records that will enable you to do so. You could, for example, make an internal note that the data wasn't quite right yet, and make that information available to the user. The user can then judge whether the absence of the data will cause the planets to halt in their orbits. This means that the software should keep track of who, what, where, how and when the user is doing things, so the situation can be modified, rectified or just plain understood at some later date. This is much more human than merely forcing the data into some arbitrary format whose correctness is judged mostly on its compliance to a file schema rather than to a human need.

Now calm down! In the real world, we accept partially and incorrectly filled-in documents all of the time. We make a mental (or otherwise) note to fix it later, and we usually do. If we forget, we fix it when we eventually discover the omission. Even if we never fix it, we somehow muddle through. Who said that values are different for computer software than it is for humans? Well, programmers, that's who. They say they are rejecting incomplete or inaccurate data for our own good, but actually they are doing it for *their* own good—so they don't have to write the more difficult code that deals with the unexpected. Humans don't die if they try to divide by zero, but stupid computer programs

do. We have a choice: We can either demand that humans *also* die or we can make our programs smarter. I know which method I choose: the one that makes the most money.

What about the—gasp!—lost data?

Yes, I realize that it is counter to everyone's wishes if information is lost. The data-entry clerk who fails to key in the invoice amount and then discards the invoice is creating a real problem. But is it really the righteous duty of the program to stop the user and point out this failure? No, it isn't. You have to consider the situation. If the application is a desktop productivity one, the user will be interacting with the program, and the results of his error will likely become apparent. In any case, the user will be driving the program like a car, and won't take kindly to having the steering wheel lock up because the stupid Chevy discovered it was low on windshield-washer fluid.

On the other hand, let's say the user is a full-time data-entry clerk keying forms into a corporate data-entry program. Our clerk does this one job for a living, and he will have spent hundreds—maybe thousands—of hours using the program. He will have a sixth sense for what is happening on the screen, and will know with a glance whether he has entered bad data, particularly if the program is using subtle, modeless visual and audible cues to keep him informed of the status of the data.

Remember, the program will be helping out: It won't demand that the user enter bounded information into unbounded gizmos. Things like part numbers that must be valid aren't going to be typed in anyway, but will be entered through a picklist of some sort. Things like addresses and phone numbers will be entered into extraction gizmos so that he can enter information more naturally. And the program will constantly give the user positive audible feedback, so the program begins to act as a partner, helping him stay aware of the status of his work.

So, how serious is the loss of data?

In a data-entry situation, a missing field can be serious, but the field will usually be entered incorrectly rather than just omitted. The program can easily help the clerk detect the problem and change it to a valid entry without stopping the proceedings. If the clerk is determined to omit necessary fields, the problem is the clerk and not the program. The percentage of clerks who fail because of either lack of ability or sociopathic tendencies is likely quite low. It

isn't the job of the data-entry program to treat all data-entry clerks as though they can't be trusted to do a simple job just because one out of a hundred can't.

Windows 95 actually offers a reasonable example of "audit, don't edit" in its installation procedure. The program is not only smart enough to adapt to unexpected situations, but it also keeps copious internal notes on its progress. If it ever crashes, it leaves behind a record of its progress up until the problem, the way a bomb-disposal expert telephones her every move to the team so that if the bomb goes off they will know what not to do next time. When the user runs the install program again, the program reads those notes and uses them to succeed. The notes tell it what it has already done successfully, so it doesn't have to do those items over. The last entry in the notes also tells it the thing that didn't work. The install program will either omit the offending task this time around or take a different tack at solving it.

Most of our information processing systems are really very tolerant of missing information. A missing name, code, number or price can almost always be reconstructed from other data in the record. If not, the data can always be reconstructed by asking the various parties involved in the transaction. Businesses do this all the time and its cost is high, but not as high as the cost of Novell's technical help lines, for example. Actually, our information processing systems can work just fine with missing data. The programmers who write these systems just don't like all of the extra work involved in dealing with missing data, so they invoke data integrity as an unbreakable, deified law, and thousands of clerks must interact with rigid fascistware to keep databases from crashing—not to prevent their business from failing.

This book isn't about worker productivity or job psychology, but it is counterproductive to treat all of your workers like idiots to protect against those few who are. It lowers everyone's productivity, encourages rapid, expensive and error-causing turnover, and decreases morale, which increases the unintentional error rate of the clerks who want to do well. It is a self-fulfilling prophecy to assume that your information workers are untrustworthy.

The moguls of the industrial age know that what I just said is true, but marginal. Oppression clearly worked well enough for them to grow and prosper, so a counter-argument can be made. However, the stereotypical role of the data-entry clerk mindlessly keypunching from stacks of paper forms while sitting in a boiler room among hundreds of identical clerks doing identical jobs is dying out very rapidly. The task of data entry is becoming less a mass-production job

and more a productivity, desktop job performed by intelligent, capable professionals and even directly by the customers. In other words, the population interacting with data-entry software is increasingly less tolerant of being treated like an unambitious, uneducated, unintelligent clerk. The imperatives of the industrial age are giving way to the imperatives of the information age, and users won't tolerate stupid software that insults them; not when they can just push a button and net surf for another few milliseconds until they find a vendor of similar goods or services that offers a software interface that treats them with respect.

Fudging

In the real world, missing information, and extra information that doesn't fit into a standard field, is an important tool for success. Information processing systems rarely handle this real-world data. They only model the rigid, repeatable data portion of transactions; a sort of skeleton of the actual transaction, which may involve dozens of meetings, travel and entertainment, names of spouses and kids, golf games and favorite sports figures. Maybe a transaction could only be completed if the termination date was extended two weeks beyond the "official" limit. Most companies would rather fudge on the termination date than see a million-dollar deal go up in smoke. In the real world, limits are fudged all of the time.

While entry systems are working to keep bad data out of the system, they almost never allow the user to fudge. There is no way to make marginal comments or to add an annotation next to a field. For example, maybe a vitally necessary item of data is missing, an interest rate, say. If the system won't allow the transaction to be entered without a valid interest rate, it stops the company from doing business. What if the interest rate field on the loan application had a penciled note next to it, initialed by the bank president, that said: "prime plus three the day the cash is delivered"? The system, working hard to maintain perfection, fails the reality test.

If the automated data-processing system is too rigid, it doesn't allow fudging. In other words, it won't model the real world. A system that rejects reality is not a good thing, even if it doesn't have any "invalid" fields. You must ask yourself the question "what is more important: Your database or your business?" The propeller-heads who manage the database and create the data-entry programs that feed it serve the CPU as master, and neither the needs of the user nor the needs of your business can overcome that. There is a significant

conflict of interest that only software design, knowledgeable in, but detached from, the development process, can resolve.

If the program is used in a professional setting and information is actually lost, this is a bad thing, but not a *big* bad thing, because it only happens very occasionally. If, however, the interface protects against losing data, it will be obnoxious to *every* entry clerk, *all* of the time, and this *is* a big bad thing. It's kind of Zen-like: if you trust your data-entry clerks, they will perform better with the increased responsibility.

Besides, the amount of lost data will be insignificant and probably recoverable. More importantly, all of the software that I have seen that had rigid validation to guarantee data integrity was as full of holes as Swiss cheese. The bottom line is that all of that data integrity stuff doesn't work against a determined invader anyway, so you might as well try a different approach.

Imagine how nice it would be if, when a user finished editing, he could request a dialog box that listed the details of suspected errors he had made, along with some suggestions as to why, and possibly some hints on fixing them. There is usually plenty of information that an auditing program can gather when it suspects problems, and most of our computers have space for it. It's a good investment.

Failing gracefully

I've already said what I think about error messages, but I'm under no illusions about the impact it will have on an industry that is chock-full of error messages. Facing reality, I accept that programs are going to issue error messages, so now I'd like to talk about how to fail with grace.

Digital computers are absolute in their behavior. They either work or they don't. Good programmers are sensitive to this nature, and tend to create programs that reflect it. When programs detect errors—really nasty internal errors—they tend to crash absolutely. Let's say a program is merrily computing along, say, downloading your email from a server, when it runs out of memory at some procedure buried deep in the internals of the program. Most of the software I know and use issues a message that says, in effect, "You are completely hosed," and then shuts down the entire program. You restart the computer only to find that the program lost your email and, when you interrogate the server, find that it has also erased your mail because it had already handed it to your program.

This is not good.

When a program discovers a fatal problem, it knows it will die. Before it goes, however, it can follow one of two strategies. It can just go ahead and crash, or it can take the time and effort to prepare for its death without hurting the user. In other words, it can go out like a disgruntled, psychotic ex-employee, taking a dozen coworkers with him in a blaze of automatic machine pistol fire, or it can tidy up its affairs, assuring that its will is complete and all of its insurance policies, bank accounts and safe deposit boxes are identified and recorded for its heirs before it peacefully goes to meet its silicon god.

Most programs are filled with data and settings. When they crash, that information is normally just discarded. The user is left holding the bag. In our email example, the program accepted email from the server—which then erased its copy—but didn't assure that the email was properly recorded locally. The email program surely didn't crash because of its own incompetence—nawwww—it was brought down by the foolishness of some irresponsible screen saver program running in the background; but the email program was the victim. No, you are the victim. If the email program had made sure that those messages were promptly written to the local disk, even before it informed the server that the messages were successfully downloaded, the problem would never have arisen, even if the stupid screen saver then crashed things.

I constantly see the unwillingness of software to dismantle itself benignly before it crashes. Even if it doesn't crash, the attitude is still there, particularly in dialog boxes. A dialog will come up and the user will enter several complex inputs and settings. On the tenth or eleventh field, the dialog rejects the user's input and shuts down the dialog. The user then calls the dialog back up, and lo, the first ten valid entries were inconsiderately discarded. Remember Mr. Jones, that incredibly mean geography teacher in high school who ripped up your entire report on South America and threw it away because you handed it in in pencil instead of ink? Why couldn't he have just asked you to transcribe it instead of forcing you to do it over? Don't you hate South America to this day? Mr. Jones could easily have been a programmer.

30

Undo

Undo is that remarkable facility that lets us reverse a previous action. Simple and elegant, the feature is of obvious value. Yet, when we examine undo from a goal-directed point of view, there appears a considerable variation in purpose and method. Undo remains important, but it's not as simple as you might think.

Assisting the exploration

Undo is the facility traditionally thought of as the rescuer of users in distress; the knight in shining armor; the cavalry galloping over the ridge; the superhero swooping in at the last second.

As a computational facility, undo has no merit. It contributes nothing to the world of computer software. Mistake-free as they are, computers have no need for undo. Human beings, on the other hand, make mistakes all of the time, and undo is a facility that exists for their exclusive use. This singular

observation should immediately tell us that of all the facilities in a program, undo should be modeled the least like its construction methods—its implementation model—and the most like the user's mental model.

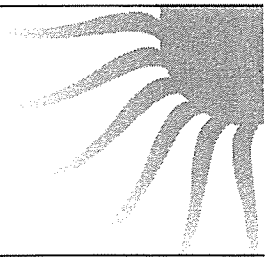
Not only do humans make mistakes, they make mistakes as part of their everyday behavior. From the standpoint of a computer, a false start, a misdirected glance, a pause, a hiccup, a sneeze, a cough, a blink, a laugh, an "uh," a "you know" are all errors. But from the standpoint of the human user, they are perfectly normal. Human mistakes are so quotidian that if you think of them as "errors" or even as abnormal behavior, you will screw up the design of your software.

The user's mental model of mistakes

Saying that the user doesn't imagine himself as making mistakes is another way of saying that his mental model doesn't include an error on his part. Following the user's mental model means absolving the user of blame. The implementation model, of course, is based on the error-free CPU. Following the implementation model means acknowledging that all culpability has to be the user's. The typical programmer normally blames the user before he blames the software. Wooop! Wooop! Wooop! Model conflict! Most software assumes that it is blameless, and any problems are purely the fault of the user.

The solution is for the user interface designer to completely abandon any shred of thought that the user could make a mistake. Users don't make mistakes in their own minds, so the program shouldn't contradict them in its user interface.

Users don't make mistakes



If we design software from the point of view that users never make mistakes, we immediately begin to see things differently. We cease to imagine the user as a module of code or a peripheral that drives the computer, and we begin to imagine him as an explorer, probing the unknown. We understand that exploration involves inevitable forays into blind alleys and box canyons, down dead ends

and into dry holes. It is natural for humans to experiment, to vary their actions, to probe gently against the veil of the unknown to see where their boundaries lie. How can he know what he can do with a tool unless he experiments with it? Of course the degree of willingness to experiment varies widely from person to person, but most people experiment at least a little bit.

From the implementation model, the programmer's point of view, such gentle, innocent probing is just a continuous series of "mistakes." From our more-enlightened, mental model point of view, his actions are natural and normal. The program has the choice of either rebuffing those perceived mistakes or assisting him in his explorations.

Undo is for exploration, not mistakes

Undo is the primary tool for supporting exploration in software user interfaces. It allows the user to reverse one or more previous actions if he decides to change his mind.

The secret to designing a successful undo system is to create it from the assumption that it supports a normal part of the everyday working set of the program's tools, avoiding any hint that undo is a tacit acknowledgment of failure by the user. The key to this is to design it so that it is less a tool for reversing errors and more one for supporting exploration. Primarily, errors are single, incorrect and unintentional actions. Exploration, by contrast, is a long series of probes and steps, some of which may be keepers and others of which need to be abandoned. Most existing undo systems treat things as single, incorrect actions, but this model is less helpful.

Undo is distastefully human

Undo is one of the more difficult exercises in practical software development. It isn't very tough algorithmically—you won't find much discussion of it in computer science textbooks—but it necessitates adding a non-trivial facility to the program and putting a lot of convoluted code into virtually every other part of it. This difficulty of construction is a big reason why undo is often omitted or implemented poorly.

Probably an equally significant barrier to the adequate implementation of undo is a psychological one: Undo is not very computer-like. Computers never make mistakes, and this is one of the programmer's main career attractions. Programmers, as a group, really appreciate the deterministic behavior of

computers. A large part of what makes programming so appealing to them is the ability to create a self-contained, self-consistent world of logical, rational behavior—a world of squared-off corners and clean-room streets. Undo, on the other hand, is all about rough edges and discards, inconsistencies and misconstrued artifacts. Undo is a human thing, not a computer thing, and because it deals with human fallibility, it represents an unpleasant and vaguely distasteful part of the programmer's job.

Undo reassures

A significant contribution that undo makes to the user is purely psychological: it reassures him. It is much easier to enter a cave if you are confident that you can get back out of it at any time. The undo function is that comforting rope ladder to the surface, supporting the user's willingness to explore further by assuring him that he can back out of any dead-end caverns.

Curiously, users often don't think about undo until they need it, in much the same way that homeowners don't think about their insurance policies until some disaster strikes. Users will frequently charge half-prepared into the cave and only start looking for the rope ladder—for undo—after they have encountered trouble.

User's mental model of undo

The user's mental model of undo is predictably variable for the simple reason that, although users need undo, it doesn't directly support a goal they bring to the task. Rather, it supports a necessary condition—trustworthiness—on the way to a real goal. It doesn't contribute positively to attaining the user's goal, but keeps some negative occurrence from spoiling the effort.

The user will visualize the undo facility in many different ways depending on the situation and his expectations. If the user is very computer-naive, he might see it as an unconditional "get-me-out-of-here" button—an escape valve or ejector-seat lever for extricating himself from a hopelessly tangled misadventure. A more experienced computer user might visualize undo as a storage facility for deleted data. A really computer-sympathetic user with a logical mind might see it as a **last-in-first-out**, or LIFO, stack of procedures that can be undone one at a time.

In order to create an effective undo facility, we must satisfy as many of these mental models as we expect our users will bring to bear.

The undo language gap

As is so common in the software industry, there is no adequate terminology to describe the types of undo that exist—they are uniformly called "undo" and left at that. This language gap contributes to the lack of innovation in new and better variants of undo. I have created a taxonomy for undo, and I show the specific names for undo variants and define them as each one is discussed.

Let's first talk about what undo operates on: the user's actions. A typical user action in a typical application has a procedure component—what the user did—and an optional data component—what information was affected. When the user requests an undo function, the procedure component of the action is reversed, and if the action had an optional data component—the user added or deleted data—that data will be deleted or added back, respectively. Cutting, pasting, drawing, typing and deleting are all actions that have a data component, so undoing them involves removing or replacing the affected text or image parts. I call functions with both a procedure and a data component **incremental actions** or **incrementals**.

Many undoable functions are dataless transformations such as a paragraph reformatting operation in a word processor or a rotation in a drawing program. Both of these operations act on data, but neither of them adds or deletes data. I call functions like these that have just a procedure component **procedure actions** or **procedurals**. Most existing undo functions don't discriminate between procedurals and incrementals but simply reverse the most recent action.

The two most-familiar types of undo in common use today are single undo and multiple undo. **Single undo** is the most basic variant, non-repeatably reversing out the effects of the most recent user action, whether procedural or incremental.

This facility is very effective because it is so simple to operate. The user interface is simple and clear, easy to describe and remember. The user gets precisely one free lunch. This is by far the most frequently implemented undo, and it is certainly adequate, if not optimal, for many programs. For some users, the absence of this simple undo is sufficient grounds to abandon a product entirely.

The user generally notices most of his command mistakes right away: something about what he did doesn't feel or look right, so he pauses to evaluate the situation. If the representation is clear, he sees his mistake and selects the undo

function to set things back to the previously correct state, then proceeds from there.

Multiple undo is repeatable—it can revert more than one previous operation, in reverse temporal order.

Normally, undo is invoked by a menu item or buttcon with an unchanging label. The user knows that triggering the idiom will undo the last operation, but there is no indication of what that operation is. I call this **blind undo**.

On the other hand, if the idiom includes a textual or visual description of the particular operation that will be undone, I call it **explanatory undo**.

If, for example, the user's last operation was to type in the word "design," an explanatory undo function on the menu would say "undo typing 'design.'" Explanatory undo is generally a much more pleasant feature than blind undo. It is fairly easy to put on a menu item, but more difficult to put on a buttcon, although putting the explanation in a ToolTip is a good compromise.

The trouble with single undo

The biggest limitation of single-level, incremental undo is when the user accidentally short-circuits the ability of the undo facility to rescue him. This problem crops up when the user doesn't notice his mistake immediately. For example, assume he deletes six paragraphs of text, then deletes one word, then decides that the six paragraphs were erroneously deleted and should be replaced. Unfortunately, performing undo now merely brings back the one word, and the six paragraphs are lost forever. The undo function has failed him by behaving literally rather than practically. Anybody can clearly see that the six paragraphs are more important than the single word, yet the program freely discarded those paragraphs in favor of the one word. The program's blindness caused it to keep a quarter and throw away a fifty-dollar bill, simply because the quarter was offered last.

In some programs any click of the mouse, however innocent of function it might be, causes the single undo function to forget the last meaningful thing the user did. This can be really frustrating if you expect undo help from the program. Although multiple undo solves these problems, it introduces some significant problems of its own.

Multiple undo

The response to the weaknesses of single-level undo has been to create a multiple-level implementation of the same, incremental undo. The program saves each action the user takes. By selecting undo repeatedly, each action can be undone in reverse order of its original invocation. In the above scenario, the user can restore the deleted word with the first invocation of undo, and can restore the precious six paragraphs with a second invocation. Having to redundantly re-delete the single word is a small price to pay for being able to recover those six valuable paragraphs. The excise of the word re-deletion tends to not be noticed the way we don't notice the cost of ambulance trips: we don't quibble over the little stuff when lives are at stake. But this doesn't change the fact that the undo mechanism is built on a faulty model, and in other circumstances, undoing functions in a strict LIFO order can make the cure as painful as the disease.

Imagine again our user deleting six paragraphs of text, then calling up another document and performing a global find-and-replace function. In order to retrieve the missing six paragraphs, the user must first unnecessarily undo the rather complex global find-and-replace operation. This time, the intervening operation was not the insignificant single-word deletion of the earlier example. The intervening operation was complex and difficult and having to undo it is clearly an unpleasant, excise effort. It would sure be nice to be able to choose which operation in the queue to undo and to be able to leave intervening—but valid—operations untouched.

Any program with undo must remember the user's last operation and, if applicable, cache any changed data. If the program implements multiple undo, it must maintain a stack of operations, the depth of which may be settable by the user as an advanced preference. Each time undo is invoked, it performs an incremental undo; reversing the most recent operation, replacing or removing data as necessary, and discarding the restored operation from the stack.

The model problems of multiple undo

The problems with multiple undo are not due to its behavior as much as to its manifest model. Most undo facilities are constructed in an unrelentingly function-centric manner. They remember what the user does function-by-function and separate the user's actions by individual function. In the time-honored way

of creating manifest models that follow implementation models, undo systems tend to model code and data structures instead of user goals. Each press of the undo button reverses precisely one function-sized bite of behavior. Reversing on a function-by-function basis is a very appropriate mental model for solving most simple problems caused by the user making an erroneous entry. Users sense it right away and fix it right away, usually within a two- or three-function limit. The new Paint program in Windows 95, for example, has a fixed, three-action undo limit. However, when the problem grows more convoluted, the incremental, multiple undo model doesn't scale up very well.

You bet your LIFO

When the user goes down a logical dead end, rather than merely mis-keying data, he can often proceed several complex steps into the unknown before realizing that he is lost and that he needs to get a bearing on known territory. At this point, however, he may have performed several interlaced functions that are not all bad. He may well want to keep some actions and nullify others, not necessarily in strict reverse order. What if the user entered some text, then edited it, then decided to undo the entry of that text but not undo the editing of it? Sort of like dividing by zero, such an operation is undefined and problematic to implement and explain. Neil Rubenking offered me this pernicious example: suppose the user did a global replace changing "tragedy" to "catastrophe," then another changing "cat" to "dog." To undo the first without the second, can the program reliably fix all of the "dogastrophes"?

In this more complex situation, the simplistic representation of the undo as a single, straight-line, LIFO stack of incrementals doesn't satisfy the way it does in simpler situations. The user may be interested in studying his actions as a menu and choosing a discontiguous subset of them for reversion, while keeping some others. This demands an explanatory undo with a more robust presentation than might otherwise be necessary for a normal, blind multiple undo. Additionally, the means for selecting from that presentation must be more sophisticated. Representing the operation in the queue to clearly show the user what he is actually undoing is a problem of some difficulty.

Redo

If you don't believe that programmers are designing our software, the redo function should convince you beyond a shadow of a doubt. By adhering rigorously to the implementation model for undo, whereby functions are literally

undone in reverse sequence, the inability to select the particular operation to undo without first undoing all of the valid intervening operations has caused the redo function to come into being. Redo essentially undoes the undo.

Redo mostly exists because it is easy to implement if the programmer has already gone to the effort to implement undo. Many programs that implement single undo treat the last undone action as an undoable action. In effect, this makes a second invocation of the undo function a redo function.

The real purpose of redo ends up being to avoid a diabolical situation in multiple undo. If the user wants to back out of a half-dozen or so operations, he presses the undo buttcon a few times, waiting to see things return to the desired state. It is very easy in this situation to press undo one time too many. He immediately sees that he has undone something desirable. Redo solves this problem by allowing him to undo the undo, putting back the last good action. Redo is really nothing more than a substitute for better visualization tools in an explanatory undo.

Special undo functions

Incremental undo

The backspace key is really an undo function, albeit a special one. When the user mis-types, the backspace key "undoes" the erroneous characters. If the user mis-types something, then enters an unrelated function such as paragraph reformatting, then presses the backspace key repeatedly, the mis-typed characters are erased and the reformatting operation is ignored. Depending on how you look at it, this can be a great flexible advantage giving the user the ability to "undo" discontiguously at any selected location, or this can be a trap for the user, as he can move the cursor and then inadvertently backspace away characters that were not the last ones keyed in.

Logic says that this latter case is a problem. Empirical observation says that it never bothered anybody. Such discontiguous incremental undo—so hard to explain in words—is so natural and easy to actually use, because everything is visible: The user can clearly see what will be "backspaced" away. Backspace is a classic example of an incremental undo, reversing only data while ignoring other, intervening actions. Yet if I described to you an undo facility that had a "pointer" that could be moved, and that undid the last function that occurred where the pointer points, you'd probably tell me that such a feature would be

patently unmanageable and would confuse the bejabbers out of a typical user. Experience tells us that backspace does nothing of the sort. It works as well as it does because its behavior is consistent with the user's mental model of the cursor: because it is the source of added characters, it can also reasonably be the locus of deleted characters.

Using this same knowledge, we could create different categories of incremental undos, like a format-undo function that would only undo preceding format commands. I call this **category-specific undo**. If the user entered some text, changed it to italic, entered some more text, increased the paragraph indentation, entered some more text, then pressed the format-undo key, only the indentation increase would be undone. A second press of the format-undo key would reverse the italicize operation. But neither invocation of the format-undo would affect the content of what the user typed in.

What are the implications of category-specific undo in a non-text program? In a graphics drawing program, for example, we could have separate undo commands for pigment application tools, transformations, and cut-and-paste. There is really no reason why we couldn't have independent undo functions for each particular class of operation in a program.

Pigment application tools include all drawing implements—pencils, pens, fills, sprayers, brushes—and all shape tools—rectangles, lines, ellipses, arrows. Transformations include all image-manipulation tools—shear, sharpness, hue, rotate, contrast, line weight. Cut-and-paste tools include all lassos, marquees, clones, drags and other repositioning tools. Unlike the backspace function in the word processor, undoing a pigment application in a draw program would be temporal and would work independent of selection. That is, the pigment that is removed first would be the last pigment applied, regardless of the current selection. In text, there is an implied order from the upper left to the lower right. Deleting from the lower right to the upper left maps to a strong, intrinsic mental model, so it seems natural. In a drawing, no such conventional order exists, so any deletion order other than one based on entry sequence would be disconcerting to the user.

A better alternative would be to undo within the current selection only. The user selects a graphic object, for example, and requests a transformation-undo. The last transformation to have been applied to that *selected object* would be reversed.

Most software users are familiar with the incremental undo and would find a category-specific undo novel and possibly disturbing. However, the ubiquitousness of the backspace key shows that incremental undo is not intrinsic to the idiom, but is, rather, a learned behavior. If more programs had modal undo tools, users would soon adapt to them. They would even come to expect them the way we expect to find the backspace key on our word processors.

Deleted data buffer

As the user works on a document for an extended time, the desire for a repository of deleted text grows. It is not that he finds the ability to incrementally undo commands useless, but, rather, that reversing actions can cease to be so function-specific. Take for example, our six missing paragraphs. If separated from us by a dozen complex formatting commands, they can be as difficult to reclaim by undoing as they are to re-key. The user is thinking "if the program would just remember the stuff I deleted and keep it in a special place, I could go get what I want directly."

What the user is imagining is a repository of the data components of his actions, rather than merely a LIFO stack of incrementals. I call this a **deleted data buffer**.

His mental model wants the missing text without regard to which function elided it. The usual manifest model forces him to not only be aware of every intermediate step but to reverse each of them in turn. To create a facility more amenable to the user, we can create, in addition to the normal undo stack, an independent buffer that collects all deleted text or data. The user can open this buffer at any time as a document and use standard cut-and-paste or click-and-drag idioms to examine and recover the desired text. If the entries in this deletion buffer were headed with simple date stamps and document names, navigation would be very simple and visual.

The user could browse the buffer of deleted data at will, randomly, rather than sequentially. Finding those six missing paragraphs would be a simple, visual procedure, regardless of the number or type of complex, intervening steps he had taken. A deleted data buffer should be offered in addition to the regular, incremental multiple undo because it complements it, and besides, the data must be saved in a buffer anyway. This feature would be quite useful in all programs, too, whether spreadsheet, drawing program, or invoice generator.

Other manifest models

The manifest model of undo in its simplest form—single—conforms to the user's mental model: *I just did something I now wish I didn't do. I want to press a button and undo that last thing I did.* Unfortunately, this manifest model rapidly diverges from the user's mental model as the complexity of the situation grows. The need for an incremental undo remains, but discerning the individual components of more than the last few operations is overkill in most cases. The user wants to back up long distances occasionally, but when he does, the granular actions will not be terrifically important to him.

Milestoning

There are yet other ways to implement undo. One of the most powerful is what I call milestoning, discussed in Chapter 8. Milestoning simply makes a copy of the entire document the way a camera taking a snapshot makes an image frozen in time. Because milestoning involves the entire document, it is always implemented by directly using the file system. The biggest difference between milestoning and other undo systems is that the user must explicitly request the milestone—the saving of the document. Once he has done this, he can proceed to safely modify the original. If he later decides that his changes were undesired, he can return to the saved copy; to a previous version.

The milestoning concept is an excellent one, and many tools exist to support it for source code. Unfortunately, no program (that I know of) supports it directly to the user. Instead, they all rely on the file system's interface, which, as we have seen, is difficult for many users to understand. If milestoning were rendered in a non-file-system user model, implementation would be quite easy, and its management would be equally simple. A single buttcon would save off the entire document in its current state. The user could save as many versions at any interval that he desires.

The step for returning to a previously milestoned version is what I call a **reversion**.

The reversion facility shown in Chapter 8 is extremely simple—too simple, actually. Its menu item merely says "Revert to Milestone," and this was sufficient for a discussion of the file system, but when considered as part of undo, it should really offer more information. Typically, it should show a list of the available saved versions of that document, along with some information about each one, like the time and day it was recorded, the name of the person who

recorded it, the length, and some optional user-entered notes. The user could choose one of these versions and the program would back down to it, discarding any intervening changes.

A relative of milestoning and reversion is a variant that I call **freezing**.

Sort of the opposite of milestoning, freezing involves locking the data in a document so that it cannot be changed. Anything that has been entered becomes unmodifiable, although new data can be added. Existing paragraphs are untouchable, but new ones can be added between older ones.

This method is much more useful on a graphic than on a text document. It is much like an artist spraying a drawing with fixative. All marks made up to that point are now permanent, yet new marks can be made at will. Images already placed on the screen are locked down and cannot be changed, but new images can be freely superimposed on the older ones. Fractal Design Painter offers a similar feature with its "wet" and "dry" paint commands.

Comparison: What would this look like?

The redo function isn't useful without the undo function, so the two have to be evaluated together to be meaningful. Besides providing robust support for the terminally indecisive, the undo-redo function is a convenient comparison tool. Say you'd like to compare the visual effect of ragged-right margins against justified right margins. Beginning with ragged-right, you invoke JUSTIFICATION. Now you press UNDO to see ragged-right and now you press REDO to see justified margins again. In effect, pressing UNDO and then REDO implements a "comparison" function; it just happens to be rendered in its implementation model. If this same function were to be added to the interface following the user's mental model, it might be manifested as a COMPARISON buttcon. This function would let you repeatedly take one step forward or backward to visually compare two states.

On my television remote control is a function labeled "Jump" (my television is a Sony; the function is present on other manufacturer's controls as well, where it has other names), which switches between the current channel and the previous channel—very convenient for viewing two programs concurrently. The jump function provides the same usefulness as the undo-redo function pair with a single command—a 50% reduction in excise for the same functionality.

When used as comparison functions, undo and redo are really one function, not two. One says "apply this change" and the other says "don't apply this change."

A single COMPARE button might more accurately represent the action to the user. Although we have been describing this tool in the context of a text-oriented word processing program, a compare function might be most useful in a graphic manipulation or drawing program, where the user is applying successive visual transformations on images. The ability to see the image *with* the transformation quickly and easily, compared to the image *without* the transformation would be a great help to the digital artist.

Doubtlessly, the compare function would remain a moderately advanced function. Just as the jump function like the one on my TV remote is probably not used by a majority of TV users, the compare button would remain one of those niceties for frequent users. This shouldn't detract from its usefulness, however, because drawing programs tend to be used very frequently by those who use them at all. For programs like this, catering to the frequent user is a reasonable design choice.

Undo is a global facility and should not be managed by local controls

We don't have individual undo functions on dialog boxes. Instead, the undo is a global, program-wide function that undoes the last action, regardless of whether it was done by direct manipulation or through a dialog box.

This makes undo problematic for embedded objects that use the OLE model. If the user makes changes to a spreadsheet embedded in a Word document, then clicks on the Word document, then invokes undo, the most recent Word action will be undone instead of the most recent spreadsheet action. I believe that users will have a difficult time with this. It fails to render the juncture between the spreadsheet and the word processing document seamlessly: the undo function ceases to be global and becomes modal. This is not an undo problem, however, but an OLE problem.

Undo-proof operations

Some operations simply cannot be undone because they involve some action that triggers a device that is not under the direct control of the program. Once an email message has been sent, for example, there is no undoing it. Once a computer has been turned off without saving data, there is no undoing it. Many operations, however, masquerade as undo-proof but are easily reversible. For example, when you save a document for the first time in most programs, it lets

you choose the name for the file. But almost no programs let you rename that file. Sure, you can "Save As..." under another name, but that just makes *another* file under the new name, leaving the old file untouched under the old name. Why? Changing the name of a file is a frequently desired "undo" feature. Because it doesn't fall into the traditional view of undo as reversing procedures one at a time, we generally don't provide an undo function for setting a filename.

Explanatory undo

Microsoft Word for Windows Version 6.0 has an unusual undo. I call it **group multiple undo**.

It is multiple-level, showing a textual description of each operation in the undo stack in a toolbar combobox. You can examine the list of past operations and select some point down in the list to undo; however, you are not undoing that one operation, but rather all operations back to that point, inclusive.

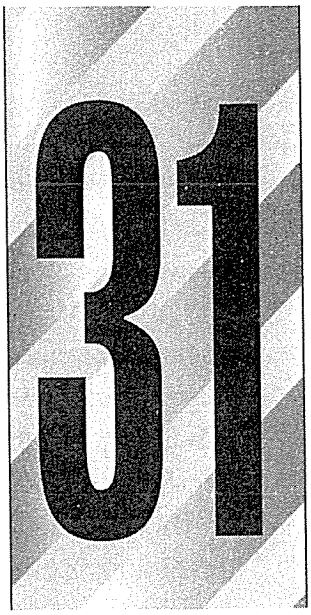
Essentially, you cannot recover the six missing paragraphs without first reversing all of the intervening operations. Once you select one or more operations to undo, the list of undone operations is now available in reverse order in the redo combobox. Redo works exactly the same way as undo works. You can select as many operations to redo as desired and all operations up to that specific one will be redone.

The program offers two visual cues to this fact. If the user selects the fifth item in the list, that item and all four items previous to it in the list are highlighted. Also, the text legend says "Undo 5 actions." The fact that they had to add that text legend tells me that, regardless of how the programmers constructed it, the users were applying a different mental model. The users imagined that they could go down the list and select a single action from the past to undo. The program didn't offer that option, so the signs were posted. This is like that door with the very pullable handle pasted with "Push" signs, and everybody still pulls on it anyway.

Part VIII: The Teacher
Education on Demand

Learning to use software should be as easy as learning the way around a new office. A little benign exploration, a couple of interesting side trips, a fortuitous meeting in the hallway; this is how we get oriented in real life. We should expect nothing less from our software. The user should be reassured at every step and generously rewarded for his curiosity. In today's information age, creativity, excitement and a sense of adventure are more important, ultimately, than correctness. There are no mistakes, only opportunities to learn.

Good at What You Do



So much emphasis has been placed on making it easy for new users to get acquainted with software that an essential point is often missed: Users spend more time as average users than they do as beginners. Software shouldn't be purposefully difficult for new users, of course, but it is even more important that the software makes everyday users powerful and satisfied. If software is part of your life or your job, it has to make you good at whatever you do.

The time users spend

Most computer users know all too well that opening the shrink-wrap on a new software product usually augurs several days of frustration and disappointment in learning the new interface. On the other hand, many experienced users of a program may find themselves continually frustrated because the program always treats them like a rank beginner. It seems impossible to find the right balance between catering to the needs of the first-timer and the needs of the expert.

Most developers are—naturally—expert users of their programs, and they tend to create interfaces that are best suited for other experts. Unfortunately, for any program, no matter how popular, there aren't going to be too many expert users out there. Prodded by complaints from customers or the marketing department, the developers add pedantic aids to the interface to lend beginners a helping hand. Unfortunately, these aids often condescend to the first-time user's ignorance. Besides, most users don't spend much time as raw beginners, so the training aids quickly turn offensive. It seems that the well-intentioned developer is cursed either way. The solution to this predicament lies in understanding the time users spend with software.

Intermediate users

Most users remain in a perpetual state of adequacy striving for fluency, with their skills ebbing and flowing like the tides, depending on how frequently they use the program. I call this a state of **perpetual intermediacy**, and such users are **perpetual intermediates**.

Imagine software users as skiers. All skiers spend time as beginners, but those who find they don't rapidly progress beyond more-falling-than-skiing quickly abandon the sport. The rest soon move off the bunny slopes onto the regular runs. Only a few ever make it onto the "double-black diamond" runs for experts. Most skiers live in the cities and only come up to the mountains a few times a year. The first trip of the season, they are very rusty, slowly and consciously recalling the little tips that they learned last year to keep themselves moving smoothly. By the last trip of the year, they have integrated all of those consciously recalled tips into their technique, so they no longer need to think about them. They are getting pretty good, boldly pushing into new territory and trying new moves. They don't have the time, money or inclination to quit their day jobs and spend all season on the slopes becoming real hot dogs, but they aren't beginners either. They are perpetual intermediates.

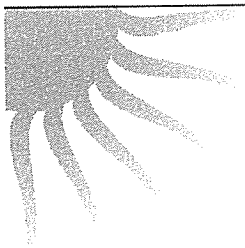
A well-rounded ski resort has a gentle slope for learning, and a few expert runs to really challenge the serious skier. But that resort also has an order of magnitude more runs for average skiers than any other. If the resort wants to stay in business, it will cater to the perpetual intermediate skier, without scaring off the beginner or insulting the expert. The beginner must find it easy to matriculate into the world of intermediacy, and the expert must not find his vertical runs obstructed by aids for bewildered perpetual intermediates.

Every software user passes through the beginner phase. Some users eventually become experts, too, but they will always be a small minority. Most users rapidly pass the beginner state but will never become experts with a particular program. Just like recreational skiers, they will spend the majority of their time as intermediate users.

A well-balanced software user interface takes the same approach as the successful ski resort. It doesn't cater to the beginner or to the expert, but rather devotes the bulk of its efforts to satisfying the perpetual intermediate. At the same time, it avoids offending either of its smaller constituencies, recognizing that they are both vital.

Most users in this middle state would like to learn more about the program, but they usually don't have the time. Occasionally, the opportunity to do so will surface. Sometimes these intermediates will use the product extensively for weeks at a time to complete a big project. During this time, they learn new things about the program. Their knowledge grows beyond its previous boundaries.

Sometimes, however, they do not use the program for months at a time and forget significant portions of what they knew. When they return to the program, they are not beginners again, but they will need reminders to jog their memory back to its former state.



Nobody wants to remain a beginner

As a percentage of hours spent with a program, beginning hours are very few, possibly less than one percent. If a user finds himself not satisfactorily progressing beyond the beginner stage after only a few hours, he will often abandon the program altogether and find another to take its place. No one is willing to remain incompetent at his job.

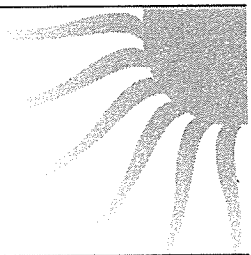
One of the most frequent mistakes committed by the design community is striving to make beginners happy. Beginnings are undeniably sensitive times, and it is easy to demoralize a first-timer, but keep in mind that the state of beginner-hood is *never* an objective. Nobody wants to remain a beginner. It is

merely a rite of passage everyone must pass through. Good software shortens that passage without bringing attention to it.

Those who can't move beyond the beginner stage soon tire of the game because there isn't much reward in it for them. The person who never gets off the bunny slope will quickly tire of skiing, so trying to make him happy there will be a waste of effort. Instead, we should determine ways to quickly hustle him out of beginner-dom into the intermediate state. Our goal should be to get him and keep him with the pack: as a perpetual intermediate.

Experts are also a vital group because they have a disproportionate influence on less-experienced users. When a prospective buyer considers your product, he will trust the expert's opinion more than an intermediate's. If the expert says "it's not very good," she may mean "it's not very good for experts," but the beginner doesn't know that and will take the expert's advice, even though it may not apply.

Perpetual intermediates usually know that advanced features exist, even though they may not need them or know how to use them. But the knowledge that they are there is reassuring to the perpetual intermediate, convincing him that he made the right choice investing in this program. The average skier may find it reassuring to know that there is a really hairy black diamond expert run just beyond those trees, even if she never intends to use it. It gives her something to aspire to and dream about.



Optimize for intermediates

Our goal is neither to pander to beginners nor to rush intermediates into expertise. Our goal is threefold: to rapidly and painlessly get beginners into intermediacy, to avoid putting obstacles in the way of those intermediates who want to become experts, and most of all, to keep perpetual intermediates happy as they stay firmly in the middle of the skill spectrum.

Command vectors

In Chapter 19, I introduced the design term "command vector" to describe different classes of control idioms. For example, menus are one command

vector, while keyboard mnemonics are another. Buttcons and direct-manipulation idioms are two other command vectors. Some command vectors offer a lot of support to new users; typically, menus and dialog boxes offer the most, which is why I call them the "pedagogic vector." Beginners will avail themselves of the pedagogy of menus as they get oriented in a new program, but perpetual intermediates often want to leave them behind to find slimmer, quicker vectors.

Head and world vectors

I describe pedagogic vectors using Don Norman's phrase "information in the world." By this, Norman means that there is a sufficiency of information available just by looking. A kiosk showing a printed map of the campus, for example, is information in the world. We don't have to bother remembering where Norman Hall is, but can find it just by reading. Opposing this is Norman's phrase "information in your head," which refers to knowledge that you have learned or memorized, like the shortcut through Alexander Hall that isn't printed on any map. Information in your head is much faster and easier to use than information in the world, but you are responsible for assuring that you learn it, that you don't forget it and that it stays up-to-date. Information in the world is slower and more cumbersome, but very dependable. A pedagogic vector is necessarily filled with information in the world, which is why I call it a **world vector**.

Conversely, keyboard commands constitute a **head vector**, because using them requires the user to have filled his head with information about the functions and their commands.

It is a mistake to impute values for these two types of command vectors. World vectors are neither better nor worse than head vectors. Either one's usefulness depends entirely on the situation. When you first moved into your neighborhood, you probably had to use a map—a world vector. After living there a couple of days, you abandoned the map because you had learned how to get home—a head vector. On the other hand, even though you know your house intimately, when you had to adjust the temperature setting on the water heater, you needed to read the instructions—a world vector—because you didn't bother to memorize them when you moved in.

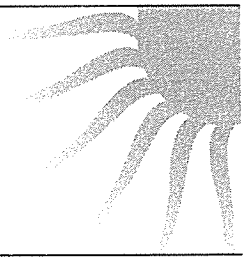
Our relationship to our software works the same way. We find ourselves easily memorizing facilities and commands that we use frequently and ignoring the details of commands that we use only rarely. This means that any operation that

is used frequently will automatically become a candidate for a head vector. After daily use, for example, we no longer really read the menus but find what we need by recognizing patterns: pull down the second menu and select the bottom-most item in the next-to-last section. We read only to verify our choice.

Because each user unintentionally memorizes commands that are used frequently, perpetual intermediates memorize a moderate subset of commands and features. This subset is called a **working set**, and the mix of commands in it is unique to each individual.

In any particular program, the working set for all perpetual intermediates will include many commands in common. In Excel, for example, most every user will enter formulas and labels, specify fonts, and print. But Sally's working set might include goal-seeking, while Elliot's working set includes linked spreadsheets. From a designer's point of view, there is no such thing as a standard working set. Although a program's mainstream functions will certainly be part of each user's working set, his individual preferences and job requirements will dictate which additional features will be included. Even custom software written for corporate operations offer a range of features from which each user will pick and choose.

Any command is a working set candidate



The commands in any person's working set will be those used frequently. The user wants those commands to be especially quick and easy to invoke. This means that the designer must provide multiple command vectors for *all* commands, because there is no way to know in advance which ones might get used over and over.

Programs that decide for the user which functions are going to be frequently used and which aren't will almost always be incorrect for a plurality of their users. The program must allow the user himself to choose his own working set.

Having said that, I'll now backpedal to say that really dangerous commands (like erase all, clear undo, abandon changes, and so on) should not have easy, parallel command vectors. Instead, they need to be protected with menus and

dialog boxes. However, if you have more than one or two dangerous commands in your program, you have way too many of them.

New users are happy with world vectors, but as they progress to become perpetual intermediates, they begin to develop their own working set and the world vectors begin to seem tedious. Each user wants to find head vectors for the contents of their working set. This is a natural and appropriate user desire and, if our software is to be judged easy-to-use, we must satisfy it. The solution consists of two components. First, we must provide a head vector in parallel to the world vector, and second, we must provide a path from the world vector to the head vector. This path, of course, is a vector itself, and I call it the **graduation vector**.

There are several ways to provide a graduation vector. The worst way is by writing it up in the program's documentation. The second worst is through the program's main online help system. These methods not only put the onus of finding the graduation vector on the user, but they leave it up to the user to realize that he needs to find it. Superior graduation vectors are either subliminally built into the interface or are at least offered in the program's interface by way of their own world vector. The latter can be implemented very easily just by adding a menu item to the standard HELP menu called SHORTCUTS. This item takes the user directly to a section of help that describes available shortcuts. This method has the benefit of being explicit and therefore pedagogic. New users can see that multiple command vectors exist and that there is an easy-to-find resource for learning them. All programs should have this shortcut item.

Design tip: Offer shortcuts from the HELP menu.

Adding subliminal graduation vectors is less problematic than it sounds. There are already two on the menus of most programs. As defined by Microsoft, a typical Windows application has two keyboard head vectors: Mnemonics and accelerators. For example, in Word, the mnemonic for SAVE is ALT-F S and the visual graduation vector for it is the underlining of the F and S in the menu name and the menu item, respectively. The accelerator for SAVE is CTRL-S. CTRL-S is noted explicitly on the right side of the menu on the same line as the SAVE item, which acts as a graduation vector.

Neither of these vectors intrudes on the new user. He may not even notice their existence until he has had the opportunity to use the program at some length—that is, until he becomes an intermediate user. Eventually, he will notice these

visual hints and will wonder about their meaning. Most reasonably intelligent people—most users—will get the accelerator connection without any help. The mnemonic vector is slightly tougher, but once the user is clued into the use of the ALT meta-key, either by direction or accident, the idiom is extremely easy to remember and use wherever it occurs.

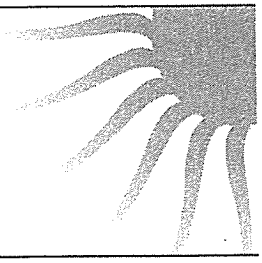
If you turn back to Figure 20-4, you can see another excellent technique where small icons are used to form a graduation vector from menus to buttcons. This repetitive use of visual images helps build the "visual fugue" I mentioned in Chapter 4. The icon identifying each function or facility should be shown on every artifact of the user interface that deals with it: each menu, each buttcon, each dialog box, every mention in the help text, every mention in the printed documentation. This graduation vector formed of visual symbols is the strongest and best technique, yet it is tragically unexploited.

ToolTips are another place where visual symbols can be used to good effect in indicating parallel command vectors. A tiny symbol—just a dot or triangle, for example—could be shown on every ToolTip that accompanied buttcons with synonymous direct-manipulation idioms.

What beginners need

Let's get one thing straight: Beginners are not stupid. As a software designer, I find it best to imagine that users are simultaneously very intelligent and very busy. They need some instruction, but not very much, and the process has to be very rapid. It can't get bogged down in training or explanation. If the ski instructor begins lecturing on meteorology and alpine ecology, he will lose his students, regardless of their aptitude for skiing. Just because a user needs to learn how to operate a program doesn't mean that he needs or wants to learn how it works inside.

Imagine users as very intelligent but very busy



On the other hand, intelligent people always learn better when they understand cause and effect, so you must give them an understanding of why things are working the way they are. It may seem like a contradiction to say they don't

have the patience for explanations but that they must understand why, and I suppose it is, but most of real life is a minestrone of contradictions. Get over it. We use mental models to bridge the contradiction. The mental model provides all the explanation the user needs without forcing him to plumb the depths of the implementation model. The user can ignore the physical functions but will need to know the scope, the benefits and the risks.

Getting beginners on board

To get beginners into a state of intermediacy requires extra help from the program, but this extra help will quickly get in their way as they become intermediates. This means that the extra help you provide must not be fixed into the interface. It must know how to go away. Generally, beginner-level assistance is designed as add-ons to the product.

This beginner information should not be permanently built into the program's interface. Our new user will either grasp the concepts and scope of the program right away or he will abandon it. He may not recall from use to use exactly which command was needed to fratz the veeblefetzer, but he will definitely remember that the veeblefetzer needs fratzing if it is consistent with his mental model.

Online help is a really bad tool for providing such add-ons. We'll talk more about help later in this chapter, but its main purpose is as a reference, and beginners don't need reference information; they need understanding. What they really need is a guided tour.

And that is precisely the way beginners should be indoctrinated: with a guide. A separate facility—a dialog box springs to mind—is a fine tool for communicating overview, scope and purpose. It doesn't necessarily have to be tightly connected to the actual controls that invoke functions because those controls already have ToolTips (they have to, to satisfy perpetual intermediates). As the user begins the program, a dialog box can appear that states the basic goals and tools of the program, naming the main features. This dialog can be quite sophisticated or simple; it doesn't really matter, as long as it stays focused on beginner issues like scope and goals and avoids perpetual intermediate and expert issues.

Beginners rely heavily on the menu to give commands. Menus may be slow and clunky, but they are thorough and wordy, so they offer reassurances. Also, the dialog boxes they usually bring up are expository and explanatory and come

with a convenient CANCEL button. The toolbar is a bit too drastic for newcomers.

What perpetual intermediates need

Perpetual intermediates need access to tools. They don't need scope and purpose explained to them because they already know these things. ToolTips are the perfect perpetual intermediate idiom. ToolTips say nothing about scope and purpose and meaning, they only state function in the briefest of idioms, consuming the least amount of video space in the process.

Perpetual intermediates know how to use reference materials. They are motivated to dig deeper and learn, as long as they don't have to tackle too much at once. This means that online help is a perpetual intermediate tool. They will use it by way of the index, so that part of help must be very comprehensive.

Perpetual intermediates will be establishing the functions that they use with regularity and those that are only used rarely. The user may experiment with obscure features, but he will soon identify—probably subconsciously—his frequently used working set. The user will demand that the tools in his working set are placed front-and-center in the user interface, easy to find and to remember.

What experts need

Experts will occasionally look for esoteric features, and they might make heavy use of a few of them. However, they primarily demand faster access to their working set of tools, which may be quite large. In other words, they want shortcuts to everything.

Any person who uses a program for hours a day will very quickly internalize the nuances of its interface. It isn't so much that users *want* to cram frequently used commands into their heads, as much as it is unavoidable. Their frequency of use both justifies and requires the memorization. If they were perpetual intermediates, they would be remembering only to forget again by the next time.

The expert is constantly, aggressively seeking to learn more and to see more connections between his actions and the program's behavior and representation. Experts appreciate new features. Their mastery of the program insulates them against becoming disturbed by the added complexity.

Beginning User	Perpetual Intermediate User	Expert User
What does the program do?	What new features are in the upgrade?	How do I automate this?
What is the program's scope?	I forgot how to import	What are the shortcuts for this command?
Where do I start?	What is this gizmo for?	Can this be changed?
How do I print?	Oops! Can I undo?	What is dangerous?
	How do I find facility X?	Is there a keyboard equivalent?
	What was the command for X?	How can I customize this?
	Remind me what *this* does	

Figure 31-1

The demands users place on software varies considerably with their experience. The tools presented to the user need to reflect this disparity. It won't be appreciated much if your program is very easy for first-timers to learn if most users are going to be perpetual intermediates. Similarly, if only professional, full-time experts will use the product, the interface needs to cater to their unique needs.

Idiosyncratically modal behavior

Many times a user population divides rather cleanly on the effectiveness of an idiom. Half of the users like one idiom while the other half strongly prefer another. I've seen development shops emotionally split on issues like this. One group becomes the "menu item" camp while the rest of the developers are the "buttcon" camp. They wrangle and argue over the relative merits of the two methods while the real answer is staring them in the face: Use both!

When a population splits like this on preferred idioms, the software designers must offer *both* idioms. Both groups must be satisfied. It is no good to satisfy one half of the population while angering the other half, regardless of which particular group you align yourself with.

I'm not saying that every individual preference needs to be accommodated. Many individuals have amusing design suggestions that should be adamantly ignored. I've heard developers state unequivocally that "dialog boxes are bad" and that "sticky menus are hateful things." These fringe voices reflect only personal taste and must be ignored. But if, in development or user testing, you find that a significant group of users has a similar preference, you must satisfy it, even if there are three or four conflicting groups. When I see this clear division of a population's preferences into two or more large groups, I say that the user's preferences are **idiosyncratically modal**.

Windows offers an excellent example of how to cater to idiosyncratically modal desires in its menu implementation. Some people like menus that work the way they do on the Macintosh: you press the mouse button on a menu bar item to make the menu appear; then—while still holding down the button—drag down the menu and release the mouse button on your choice. Other people find this procedure difficult and prefer a way to accomplish it without having to awkwardly hold the mouse button down while they drag. Windows neatly satisfies this by letting the user click-and-release on the menu bar item to make the menu appear. Then the user can move the mouse—button released—to the menu item of his choice. Another click-and-release selects the item and closes the menu. Just as on the Macintosh interface, the user can still click-and-drag to select a menu item. The brilliance of these idioms is that they coexist quite peacefully with each other. Any user can freely intermix the two idioms, or stick consistently with one or the other and the program requires no change. There are no preferences or options to be set; it just works.

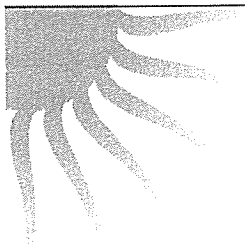
In Windows 95, Microsoft has added a third idiosyncratically modal idiom to the menu behavior with their new "menu mode." The user clicks-and-releases as before, but now he can drag the mouse along the menu bar and the other menus are triggered in turn. Amazingly, now all three idioms are accommodated seamlessly and integrally. It's as though the lion lay down with the lamb *and* the hyena.

If you are writing a transient posture application that won't be used frequently, go ahead and provide just one command vector. If you are creating a sovereign application that will be used frequently by a broad spectrum of users, however, you must expect idiosyncratically modal behavior, along with very individual working sets. The only solution is multiple command vectors for all functions.

Commensurate effort

There is a principle that grants some measure of relief to overburdened interface designers. I call it the principle of **commensurate effort** and although it applies to all users, it is particularly pertinent to perpetual intermediates and experts.

The principle merely states that people will willingly work harder for something that is more valuable to get. If I really want something, I will work hard to get it. The catch, of course, is that value is in the eye of the beholder. It has nothing to do with how technically difficult a feature was to implement. If a person wants to become a good tennis player, for example, he will get out on the court and play very hard. To someone who doesn't like tennis, any amount of the sport is tedious effort. If a user really wants to format beautiful documents with multiple columns, several fonts and fancy headings, he will be highly motivated to explore the recesses of the program to learn how. He will be putting commensurate effort into the project. If some other user just wants to print plain old documents in one column and one font, no amount of inducement will get him to learn those more-advanced formatting features.



Users make commensurate effort

This means that if you add features to your program that are necessarily complex to manage, users will be willing to tolerate that complexity only if the rewards are worth it. This is why a program's user interface can't be complex if it's being used to achieve simple results. Of course, as users get more experienced with the feature, they search for shortcuts, and you must provide them. When software follows commensurate effort, the learning curve doesn't go away, but it disappears from the user's mind, which is just as good.

The typers versus the pointers

Beginners like to point gingerly with the mouse the way they might dip their toe in the surf. Frequent users like to remember and use keyboard idioms that speed up their work. Everybody uses both and, surprisingly, many people tend

to use an unorthodox combination of keyboard and mouse idioms to select and manipulate information in Windows. The truth is that we use both interchangeably; and often within a single object-verb operation. This situation was not well understood by the GUI inventors, and it was summarily rejected by the original Macintosh, though they eventually added arrow keys in deference to it.

IBM's PC was very influential in the '80s, dictating many of our still-extant standards. IBM made the indefensible argument that the mouse wasn't necessary; an argument equivalent to saying that the rear view mirrors on your car aren't necessary for driving. Strictly true, but counterproductive. I can't prove it, but I suspect that IBM thought this whole GUI thing was just a passing fad and were unwilling to commit to it. Even after events proved them wrong, IBM remained ambivalent about the mouse, refusing to adopt it wholeheartedly in both their hardware and their software.

CUA

IBM's ambivalence affected the development of their user interface standard for all IBM software, which was called Common User Access, or CUA. Notably, CUA included both the mouse and the keyboard as co-equal command vectors. Back when Microsoft was pals with IBM, Windows was mandated to be CUA compliant (about the time of Windows 3.0). The CUA requirement was obnoxious because IBM was unable to define it precisely or completely, yet their world-class bureaucracy was very skilled at assuring that it wasn't ignored. As soon as Bill Gates and IBM ceased their intimate relationship, most talk of CUA was abandoned at Microsoft. However, the seeds had been sown, the advantages made visible, and Windows today wisely supports the notion that users may wish to use the keyboard or the mouse in any mix and that well-behaved Windows applications will support them.

Apple was the opposite of IBM, very sympathetic to the beginning computer user. They envisioned touch-typists as corporate-drones and thumbed their noses at them. Apple treated the keyboard as a significantly less important and less useful command vector, adamantly refusing to put special function keys and arrow keys on it for several years. This choice was as silly in its way as IBM's insistence that software work totally without mice.

The argument between the Macintosh "pointers" and the PC "typers" has raged loudly and inconclusively for many years. The truth, as might be expected, eluded both camps and involves compromise.

The big problem with CUA is its demand that the keyboard must be capable of doing *everything;* that computers will normally not be equipped with a mouse. This demand and the expectation it is based on have been proven incorrect. Today, the mouse is omnipresent, as is the keyboard. Many tasks are very easy and natural with a mouse and very difficult and unnatural with the keyboard. For example, moving a pluralized window from one position on the desktop to another is trivial with the mouse but demands a tortuous evolution when done with the keyboard. I don't know of anybody who moves windows around with the keyboard, do you?

CUA's all-or-nothing attitude is wrong and the omnipresence of mice can now be taken for granted. Operations that are simple and direct with a mouse and complex and indirect with the keyboard can safely be supported only by the mouse idiom. It is a waste of time to implement them with the keyboard.

Having said that, I should emphasize that such idioms are not common, and this shouldn't be used as an excuse for dropping keyboard support of actions that are not clearly better in every respect with the mouse.

The diamond

In the '70s and early '80s, computer users mastered either CP/M or DOS, both command-line operating systems. The programs that ran on these operating systems were also structured with command-line interfaces, and the learning curve for these programs was steep indeed. However, the rewards were great for undergoing the initial torture of learning. Now, I'm not advocating a return to WordStar or VisiCalc, but there was much of value in those old programs that shouldn't be tossed out indiscriminately.

Without a mouse, those old programs could only provide head vectors—the user had to memorize all commands—on the only input device commonly available at the time: the keyboard. Even though beginners couldn't be accommodated with world vectors, the perpetual intermediates and experts were happy with the high-productivity keyboard command vectors and they were willing—commensurate effort, remember—to memorize them if they had to. Software designers took full advantage of the ten (later twelve) special function keys that IBM made a standard. They also made extensive and effective use of the two meta-keys, CTRL and ALT.

WordStar, in particular, had a powerful influence over microcomputer users who were touch-typists. By holding down the CTRL key with your left pinkie

and simultaneously pressing a letter key with your left fore- or middle-finger you could navigate your document with incredible speed and ease. That is, *if* you were a touch-typist and *after* you memorized the idiom. WordStar called it "the diamond" because the salient keys were arranged in the slightly mnemonic diamond shape, shown in Figure 31-2. Beginners and hunt-and-peck typists *hated* WordStar. I used WordStar (and its predecessor, WordMaster) for at least a dozen years. My fingers still remember that diamond even though I couldn't consciously tell you any of the commands. I still find myself fruitlessly typing a CTRL-SOMETHING to move up and down in Word and Excel.

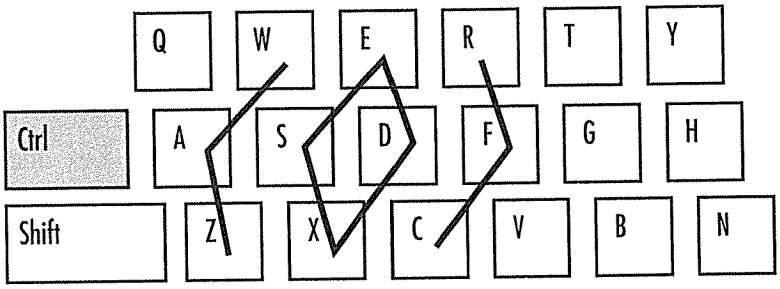


Figure 31-2

The basic diamond in WordStar was the E, S, D and X. You held the Ctrl key down with your pinkie, then pressed the E to move the cursor up one line, the X to move down one line, the S to move left one character, the D to move right one character. Ctrl-A moved the cursor left one word, and Ctrl-F moved right one word. Ctrl-W scrolled the screen up one line, and Ctrl-Z scrolled the screen down one line. Ctrl-R scrolled the screen up one page, and Ctrl-C scrolled it down one page. The diamond could be mastered in a few hours of practice; it was and still is the fastest way to move around in a document if you are a touch-typist. I blame IBM for destroying this wonderful idiom. They moved the Ctrl key away from the pinkie position, and forced all navigation functions onto the ten-key pad to the right of the regular keyboard. Now you have a Hobson's choice between two typist-hostile navigation methods: IBM's ten-key or Apple's mouse.

WordStar owned some ninety percent of the word processor market in the late-seventies and early-eighties, but in the late-eighties lost virtually all of its clientele to the GUI-based word processors like Microsoft Word, WordPerfect and Lotus's Ami Pro. These new products made it easy for beginners, but unfortunately, the baby got tossed out with the bath water. You can point-and-click in Word, and there are parallel keyboard commands for navigating, but they are all based on the numeric keypad and require you to move your right hand away from the home row to use them. This doesn't sit well with touch-typists, nor with me. I'm a reasonably skilled touch-typist (thanks, Dad, for insisting I learn when I was 13 years old!) and I would still like to use that old WordStar

diamond in all of my contemporary programs. I don't want to return to the days when the diamond was forcibly inflicted on users, but perpetual intermediates and experts can still benefit from such high-speed, high-productivity command vectors, despite their high demands.

Standards

It has been said that the great scientific disciplines are examples of giants standing on the shoulders of other giants. It has also been said that the software industry is an example of midgets standing on the toes of other midgets. This old joke seems particularly relevant with respect to standards.

Nothing is more paradoxical in the software industry than standards. Standards are easily the biggest aid to interface designers. Standards are also the biggest obstacle for user interface designers. The lack of standards is the biggest aid to interface designers. The lack of standards is also the biggest obstacle for user interface designers.

If this were the business of making soap or paper or drinking glasses, we would have well-established standards and they would be slowly, so slowly creeping forward decade by decade. The solidity of standards allows producers and consumers alike to depend on quality, features and price. There are no great technological changes in the world of soap or paper or drinking glasses, so their respective apple-carts remain upright. In software, though, new technologies and techniques appear monthly. In the world of user interface design, we desperately need all the advances we can get. Naturally—it seems—we should immediately discard the old for the new, except that users and vendors hate moving targets.

Standards allow us to consolidate our technological gains and exploit our collective accomplishments. Standards are a plateau in the steep climb of invention. They may hinder innovation, but they allow commercial products to build a base of customers. This commercial success in turn funds more invention.

The big standards, like interface paradigms and platforms, are set de facto—by the market. Only now, after twenty years, is Microsoft powerful enough to establish standards de jure—by declaration. And even there, Microsoft is only effective on the fringe standards like MAPI and TAPI, while standards on center stage like ODBC and OLE are fiercely contended by others.

IBM's foolish CUA was a last-gasp attempt by the fading giant to set a standard de jure in an area where none was wanted; plus, they had virtually no expertise.

CUA failed because it was too much, too soon and too much of a lowest common denominator. Calculated to offend no one, it offended all.

On the other hand, designers and inventors hate standards because they cramp their style and obstruct their pushing out of the boundaries of the envelope.

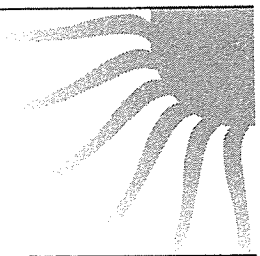
I hear amateur user interface designers and programmers speak of user interface standards all the time as though they are codicils recorded in a big book somewhere. It's though Apple or Microsoft had figured out the right methods for all time and it is our duty to perpetuate them. Sure, both companies actually have published user interface guidelines, but both companies freely break them and then update the guidelines.

In the Windows world, we really don't have standards as much as we have big examples. Our standards are "what Excel does" or "what Word does." Every time Microsoft proposes something as standard, they willfully go ahead and change it for something better in the next version. And they should. Interface design is today in its infancy and it is silly to think that there is benefit in chiseling our baby steps in granite.

Windows 1.0 violated the standards set by MS-DOS. Windows 2.0 violated the standards of 1.0. Windows 3.0, 3.1 and Windows 95 each stepped on their predecessors in turn. The Macintosh was such a spectacular achievement because it absolutely abandoned all of Apple's previous platforms.

Conversely, much of the strength of the Mac came from the fact that vendors all followed Apple's lead and made their interfaces look, work and act alike. Similarly, many successful Windows programs have been unabashedly modeled after Word or Excel. You should be getting a sense for where I'm going by now, as I keep goring user interface design on the horns of the "standard" dilemma. When we find two otherwise true propositions in direct opposition, the answer is to rephrase the question. Instead of asking whether we should follow standards, the question must be articulated more like this: When should we *violate* standards? The answer to this question is simple, but difficult: We should violate standards whenever we have a darn good reason to.

Obey standards unless you've got a darn good reason



Of course, this begs the question: What makes a darn good reason? Personal preference is out, including those of your client or user-test subjects. I'd like to answer "when a new idiom is measurably better," but I am deeply suspicious of the objectivity of contemporary "measurement" approaches. The real answer is the de facto answer: when the idiom can be seen to be manifestly better, go ahead and use it. This is how the toolbar came into existence, along with buttcons, outlines, tabbed dialogs and many other idioms. Scientists and academics may have been examining these artifacts in their labs, but it was their presence in real-world software that showed the way. Your reason may ultimately prove to *not* be darn good and your product will suffer—possibly die—but designers will learn from its lack of merit. This is what Christopher Alexander, in *Notes on the Synthesis of Form* (Harvard University Press, 1964), calls the "unselfconscious process," an indigenous and unexamined process of slow and tiny forward increments as individuals attempt to improve solutions. Like art.

Online help

Online help is not a part of good user interface design. Online help is really the same as, and not significantly better than, printed documentation, and in many ways it is a lot worse. Both forms of documentation are largely useless for anything other than a reference tool for perpetual intermediates. Ultimately, online help is not important, the way that the user manual of your car is not important. If you find yourself needing it, it means that your car is badly designed. The design is what is important.

A complex program with many features and functions should come with a reference document: a place where users who wish to expand their horizons with a product can find definitive answers. This document can be a printed manual or it can be online help. The printed manual is comfortable, browsable and friendly and can be carried around. The online help is searchable, semi-comfortable, very lightweight and cheap. Online help could be improved somewhat.

The index

Because you don't read a manual like a novel, the key to a successful and effective reference document is the quality of the tools for finding what you want in it. Essentially, this means its index. A printed manual has an index in the back that you use manually. Online help has an automatic index search facility. It is

often said by the defenders of online help that the automatic search facility is more powerful than a manual, hardcopy version. This is theoretically true, but not relevant. The difference between a good index and a bad index is the quality of its entries and not of its search tools.

I suspect that few if any of the online help facilities I've seen were indexed by a professional indexer. Certainly, I can never find what I need in the help systems I've used, and I can find dozens of entries whose presence would seem totally reasonable and obvious that are missing from any help system. Even the help system in Word, for example, which has a really superb index, offered up these omissions in about five minutes of random probing: *lost, can't find, video, sound, report, backspace, escape, enter, page down, page up.* However many entries are in your program's index, you could probably benefit from doubling the number.

What's more, the index has to be generated by examining the program and all of its features, not by examining the help text. This is not easy, because it demands that a highly skilled indexer be intimately familiar with all of the features of the program. I suspect it's easier to rework the interface to improve it than to create a really good index.

The list of index entries is arguably more important than the text of the entries themselves. The user will forgive a poorly written entry with more alacrity than he will forgive a missing entry. The index must have as many synonyms as possible for topics. Prepare for it to be huge. The user who needs to solve a problem will be thinking "how do I turn this cell black," not "how can I set the *shading* of this cell to 100%." If the entry is listed under shading, the index fails the user. The more goal-directed your thinking is, the better the index will map to what might possibly pop into the user's head when he is looking something up. The index model that I like is the one in *The Joy of Cooking*, by Irma S. Rombaur & Marion Rombaur Becker (Bobbs-Merrill, 1962). That index is one of the most complete and robust of any I've used.

Shortcuts

One of the features missing from every help system I've seen is an option I call "Shortcuts." In my designs, I place this option prominently on the help menu. It is a graduation vector, showing in digest form all of the head vectors for the program's various features. It is a very necessary component on any online help system because it provides what perpetual intermediates need the most: access

to features. They need the tools and commands more than they need detailed instructions.

The other missing ingredient from online help systems is overview. I want to know how the "Enter Macro" command works, and the help system explains uselessly that it is the facility that lets me enter macros into the system. What I need to know is scope, effect, power, up-side, down-side and why I might want to use this facility both in absolute terms and in comparison to similar products from other vendors.

Not for beginners

I see many help systems that assume that their role is to provide assistance to beginners. This is not true. Beginners stay away from the help system because it is generally just as complex as the program. Besides, any program whose basic functioning is too hard to figure out just by experimentation is unacceptably bad, and no amount of help text will resurrect it. Online help should ignore first-time users and concentrate on those people who are already successfully using the product, but who want to expand their horizons: the perpetual intermediates.

About boxes are an excellent idiom for offering basic assistance to users, though it is rarely done.

Better help

ToolTips are modeless online help, and they are incredibly effective. Why can't we have more idioms like these? Our "standard" help systems are implemented in a separate program that covers up most of the program for which it is offering help. If I asked a human about some feature, chances are excellent that he would use his finger to point to something on the screen to augment his explanation. A separate help program that obscures the main program cannot do this. The basic form of current online help systems is weak. I'd rather see the help presented transparently over the program or built right into the face of it.

Wizards

Wizards are a new idiom unleashed on the world by Microsoft, and they are rapidly gaining popularity among programmers and erstwhile user interface designers. I have big reservations about their popularity among users.

Basically, a wizard is a series of dialogs that attempt to guarantee success in using a feature by stepping the user through a series of dialog boxes. These dialogs parallel a complex procedure that is "normally" used to manage a feature of the program. For example, a wizard helps the user create a presentation in PowerPoint 4.0.

Programmers really like wizards because they get to treat the user like a peripheral device. Each of the wizard's dialogs asks the user a question or two, and in the end the program performs whatever task was requested. They are a fine example of interrogation tactics on the program's part.

Wizards are written as step-by-step procedures, rather than as informed conversations between user and program. The user tends to feel like the conductor of a robot orchestra: swinging the baton to set the pace, but otherwise having no influence on the proceedings. In this way, wizards rapidly devolve into exercises in confirmation messaging. The user learns that he merely presses the NEXT button on each screen without critically analyzing why.

There is a place for wizards in actions that are very rarely used, like installation or deinstallation. They are too demeaning to the user to be used in daily services, though.

A better way to create a wizard is to make a simple, automatic function that asks no questions of the user but that just goes off and does the job. If it creates a presentation, for example, it should create it, and then let the user have the option, using standard tools, to later change the presentation. The interrogation tactics of the typical wizard are not friendly, reassuring or particularly helpful. The wizard often doesn't explain to the user what is going on.

I'd much rather see the effort that is going into wizards go into designing better user interfaces. Wizards are having the opposite effect, though. They are giving programmers license to put raw implementation model interfaces on complex features with the bland assurance that "we'll make it easy with a wizard." This is all too reminiscent of the "we'll be sure to document it well in the manual" cry of years past.

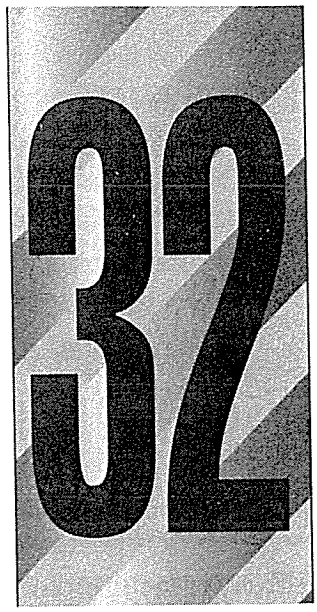
The inverted meta-question

In Chapter 13, I introduced the concept of the meta-question, where the user must ask the program for permission to ask a question. This is pure excise and

should never be used, but its opposite can be useful in certain circumstances. I call this the **inverted meta-question**.

Instead of forcing the user to ask to ask, the inverted meta-question tells a dialog to go away and not ask again. In this way, a user can make an unhelpful dialog box stop badgering him, even though the program mistakenly thinks it is helping. For example, every time I ask my drawing program to insert clipart, it asks me what directory to search in. It would be great if I could check a box on that dialog telling it to never come back again. From now on, it would only look for clipart in the last directory, which is always correct. If I were to change the directory, I could get the dialog back by requesting the function in a special way, such as with a meta-key. If a beginner inadvertently dismisses a dialog box and can't figure out how to get it back, he may benefit from an easy-to-identify safety net idiom in some prominent place. A help menu item saying "Bring back all dismissed dialogs," for example, may be just the ticket.

We need to spend more time making our programs powerful and easy to use for perpetual intermediate users. We must accommodate beginners and experts, too, but not to the discomfort of the largest segment of users.



Installation, Configuration and Personalization

Most user interface designers find themselves facing a conundrum regarding whether to make their products user-customizable. It is easy to be torn between the user's need to have things done his way, and the clear problem this creates when the program becomes hard to navigate because familiar elements of it are either gone or moved. The solution is not to choose one or the other of these options but to cast the problem in a different light.

Navigation is by reference to permanent objects

The user must be able to **navigate** through the features and facilities of a complex and powerful program. He must be able to stay oriented in the program as he moves from screen to screen.

A user can navigate a program if he always understands what he has to do next, knows what state the program is in and knows how to find the tools he needs.

One of the most important aids to navigation is a simple interface without a lot of places to navigate to. By places, I mean modes, forms and major dialogs. If the number of modes is kept to a minimum (like one or two), the user's ability to stay oriented in them increases dramatically. Similarly, if the number of forms or views is never more than one or two, the user can make sense out of them. Programs with thirty, forty or more forms are not navigable under any circumstances.

Beyond reducing the number of navigable places, the only way to enhance the user's ability to find his way around in the program is by providing better points of reference. In the same way that sailors navigate by reference to shorelines or stars, users navigate by reference to **permanent objects** placed in the program's user interface.

These permanent objects, in a GUI world, always include the program's windows. Each program will most likely have a main, top-level window. Also considered permanent objects are the salient features of that window: things like menu-bars, toolbars, and other palettes or visual features like status bars and rulers. Generally, each window of the program has a distinctive look that will soon become instantly recognizable.

Depending on the application, the contents of the client area of the program's main window will also be easily and permanently recognizable. Some programs may offer a few different views of their data, so the overall aspect of their screen will change depending on the view chosen. Generally though, a program's distinctive look will come from its unique combination of menus, palettes and toolbars. This means that menus and toolbars must be considered permanent objects and aids to navigation. You don't need a lot of permanent objects to navigate successfully. They just need to be visible. Needless to say, permanent objects can't aid navigation if they are removed.

On the other hand, people like to change things around to suit themselves. Even beginners, not to mention perpetual intermediates, like to put their own personal stamp on a program, changing it so that it looks or acts uniquely their own. You can see this in any office, where, even though everyone has an identical little gray cubicle, you can tell them apart by the pictures of spouses and kids, plants, favorite paintings or quotes, and Dilbert cartoons.

Actually, those pictures and cartoons are serious aids to navigation because, although they may not be permanent objects themselves, they are decorations placed on permanent objects. Decorating the permanent objects—the walls—

gives them individuality without removing them. It allows you to recognize one hallway as being different from dozens of identical hallways because it is the one with the big poster of Christie Brinkley, Brad Pitt or Opus. I use the term **personalization** to describe the decoration of permanent objects.

Personalization makes the places in which we work more likable and familiar. It makes them more human and pleasant to be in. The same is true of software, and giving the user the ability to decorate his personal program is good, both for fun and for practical purpose.

On the other hand, actually moving a permanent object around can really hamper navigation. If the facilities people come into your office over the weekend and rearrange all of the cubicles, Dilbert cartoons notwithstanding, finding your office again on Monday morning will be tough.

Is this an apparent contradiction? Not really. Adding decoration to permanent objects helps navigation, while moving the permanent objects hinders navigation. I use the term **configuration** to describe adding, moving or deleting permanent objects.

Configuration is desirable for experienced users. Perpetual intermediates, when they have established a working set of functions, will want to configure the interface to make those functions easier to find and use. They will also want to tune the program itself for speed and ease. They won't do excessive configuration, but they will want to make a couple of changes, and the ability to do so can make the difference between liking the program and not liking it.

Configuration is a necessity for expert users. They are already well beyond the need for more traditional navigation aids because they are so familiar with the product. Experts may use the program for several hours every day; in fact, it is probably their main application for accomplishing the bulk of their job, the way Word is my constant companion as I write this book. I have gone ahead and configured the toolbar for my version of Word so that it fits my style of working. There are cryptic-looking tools present that only I know about, and I have removed some standard tools that other users might expect.

What are typical permanent objects

The most prominent permanent object in a program is the main window and its caption and menu bars. Part of the pedagogic benefit of the menu comes from its reliability and consistency. Unexpected changes to a program's menus

can deeply reduce the user's trust in them. This is true for menu items as well as for individual menus. It is okay to add items to the bottom of a menu, but the standard suite of items in the main part of it should change only for a clearly demonstrable need.

If the program has a toolbar, it should also be considered a recognizable permanent object. Because toolbars are idioms for perpetual intermediates rather than for beginners, the strictures against changing menu items don't apply quite as strongly to individual buttcons. Removing the toolbar itself is certainly a significant motion of a permanent object, and, although the ability to do so should be there, it shouldn't be offered casually, and the user should be protected against accidentally triggering it. I've seen programs that had buttcons on the toolbar that made the toolbar disappear! This is completely inappropriate.

Status bars, tool palettes and fixed areas of the screen where data is displayed or edited should also be considered permanent objects that cannot be changed without penalty.

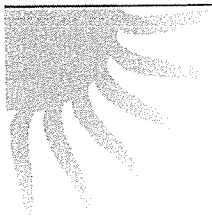
Pull at your own risk

Configuring software can be dangerous. It is a double-edged sword, offering power and flexibility, but demanding in turn that its user understand the potential danger of his actions. Command vectors that control configuration are what I call **ejector seat levers**.

In the cockpit of every jet fighter is a brightly painted lever that, when pulled, fires a small rocket engine underneath the pilot's seat, blowing the pilot, seat and all, out of the aircraft to parachute safely to earth. Ejector seat levers can only be used once, and their consequences are catastrophic. The jet will be destroyed, the pilot may be injured by the sudden and violent acceleration, and, depending on where and how she lands, she may not even survive. On the other hand, a damaged, out-of-control jet airplane headed for certain disaster is not a healthy place for pilots to remain either.

Just like a jet fighter needs an ejector seat lever, complex business programs need configuration facilities. The vagaries of business and the demands placed on the software force it to adapt to specific situations, and it had better be able to do so. Companies that pay millions of dollars for custom software or site licenses for thousands of copies of shrink-wrapped products will not take kindly to a program's inability to adapt to the way things are done in that particular

company. The program must adapt, but such adaptation can be considered a one-time procedure, or something done only by the corporate IS staff on very rare occasions. In other words, ejector seat levers may need to be used, but they shouldn't be used very often. After all, the pilot's seat is normally considered a permanent object in the airplane.



Hide the ejector seat levers

Programs must have ejector seat levers so that users can—occasionally—move permanent objects around. But the one thing that must never happen is accidental deployment of the ejector seat. See Figure 32-1. The interface design must assure that the user can never inadvertently fire the ejector seat when all he wants to do is make some minor adjustment to the program.

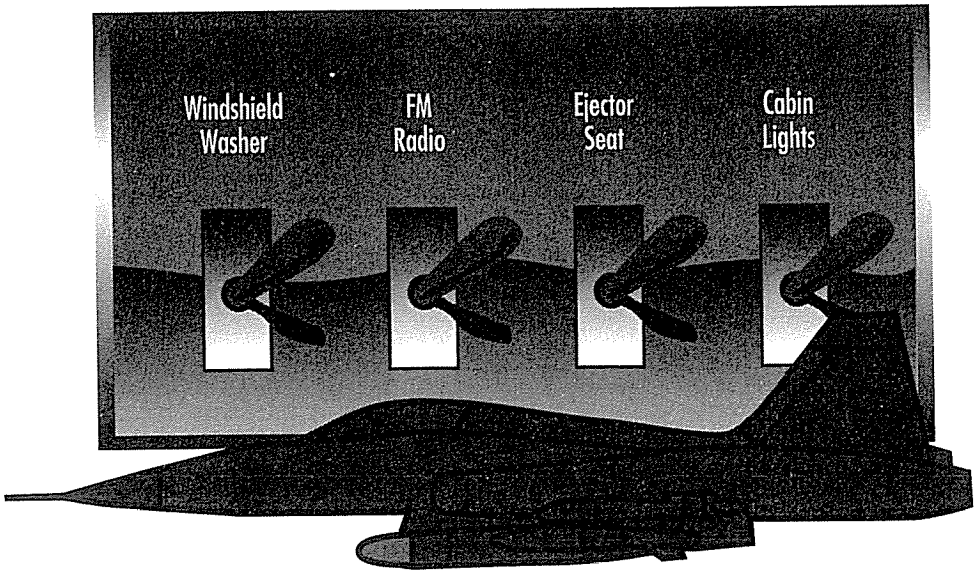


Figure 32-1

Ejector seat levers have catastrophic results. One minute, the pilot is safely ensconced in her jet, and the next she is tumbling end-over-end in the wild blue yonder while her jet goes on without her. The ejector seat is necessary for the pilot's safety, but a lot of design work has gone into assuring that it never gets fired inadvertently. Allowing an unsuspecting user to configure a program by changing permanent objects is comparable to firing the ejection seat by accident. Hide those ejector seat levers!

Ejector seat levers come in two basic varieties: those that cause a large visual dislocation in the program and those that perform some irreversible action. Both of these functions should be hidden from inexperienced users. In PowerPoint, for example, you can switch views between drawing, outline and slide sorter. Going between any of these, particularly from drawing to outline, can be shocking to a user who is unfamiliar with the feature. If the user is in draw mode and accidentally presses the outline button, for example, all of those nice graphic images that he has been working on immediately disappear to be replaced by a list of the slide text. Both views are excellent tools, and frequent switching between them is normal for perpetual intermediate users. We don't want to remove this function—actually we want to make it quite prominent and easy to reach; we do, however, want to assure that it doesn't get used inadvertently. Microsoft has solved this problem by putting buttcons that control changes in the view at the bottom of the screen—on the left end of the horizontal scroll-bar—and kept them off of the main toolbar at the top of the screen, where new users are more likely to experiment. A first-timer may find the view buttcons at the bottom of the screen, but their segregated presence clearly, and visually, indicates that they are special.

Moving buttcons around on the toolbar is a form of personalization. However, the leftmost three buttcons on many programs, which correspond to File New, File Open and File Save, are now so common that they can be considered permanent objects. A user who moves these around is configuring his program as much as he is personalizing it. Thus, you can see that there is a gray boundary between configuration and personalization.

Changing the color of objects on the screen is clearly a personalization chore, and one that is appreciated by many users. Windows has always been very accommodating in this respect, allowing users to independently change the color of each component of the windows interface, including the color and pattern of the desktop itself. Windows 95 finally gives users a *practical* ability to change the system font, too. Personalization is one of those idiosyncratically modal things. People either like it or don't. I have shown new users how to configure their screen's colors only to have them plead with me to "put it back" as though I were dangling them out the door of an airplane. I also know users who change every color on the screen as their first task when installing a new system. You must accommodate both of these categories of users.

Personalization isn't just a system-wide activity. You can add the ability to do it to your own program. It is easy to find opportunities for personalization, just

ask around your company. People will say things like "I think this should be over here" or "I don't like this shade of yellow." These are good indicators of areas for allowing personalization in your application.

The tools for personalizing must be simple and easy to use, giving the user a visual preview of their selections. Above all, they must be easy to undo. A dialog box that lets users change colors should offer a function that returns everything to the factory settings.

Users really like personalization. It allows them to feel part of the computing process; to buy into the task being performed. If you want users to like your program, offer them little perquisites like personalization, and they will really appreciate it. A user that appreciates your program will tell his friends, and, as marketing people know, there is no more convincing form of advertising than word of mouth.

Conversely, most users won't squawk if they can't configure your program as long as it does its job well. Some really expert users may feel slighted, but they will still use and appreciate your program if it works the way they expect.

Corporate MIS managers really like configuration. It allows them to subtly coerce corporate users into using common methods. The MIS manager (or the equivalent person whose task it is to purchase and configure PC software for the entire corporation) will really appreciate the ability to add special macros and commands to menus and toolbars that make the off-the-shelf software work more intimately with established company products and standards. An MIS manager who appreciates your program will tell his friends, and, as marketing people know, there is no more convincing form of advertising than word of mouth.

Many companies base their buying decisions on the configurability of programs. They figure that they are buying ten or twenty thousand copies of a program and they should be able to adapt it to their particular style of work. It is not a whim of the programming staff that has made the programs in Microsoft Office among the most configurable shrink-wrapped applications available.

The corporate look

Following visual standards is great, but this doesn't mean that your program has to have the exact, pixel-for-pixel look of the big guys. There is a lot to be said for establishing your own identity, your own look.

The same way that an individual user improves navigation by personalizing his program, as a software publisher, you can personalize your entire application. By putting identifying marks on all of the components of the program, you help in creating a branded product, one that the user subconsciously recognizes and imputes value to.

If all of the windows in your program, particularly all of the dialog boxes, have a consistent family appearance, the user can navigate more easily among the welter of windows on the screen. I'm talking only about decoration here, not behavioral divergence. All of the dialog boxes in Microsoft's Office suite, for example, use the 3-D, conservative gray, corporate etched look. Borland uses the 3-D, mottled-steel, big-bright-bit-mapped button look. Many users can identify the publisher of a program from just a glance at a dialog box. Unfortunately, many application developers think this means that their dialogs should look exactly, pixel-for-pixel, like a Microsoft, Lotus or Borland dialog box.

I'm not sure why they believe this. Most companies want to establish a proprietary look in all other corporate artifacts, so why would the boss insist on such slavish similarity on the screen? If every envelope, sheet of paper and business card has the company logo printed prominently on it, why shouldn't it also be present on each dialog box? Admittedly, it seems that logos consume pixels only to serve the publisher, but there are some collateral benefits for the user in better navigation and trust.

Users *like* brands. They buy Kleenex, Nike and Ralph Lauren because it lets them feel they are part of an exclusive club. Brands indicate quality in the product and discrimination and taste in the user. Indicating your company brand on a product can reassure users. The visual recognition afforded by logos can also help users find their way around a crowded screen.

There are several easy ways to create standard, yet visually unique, dialog boxes. You can change the background color of all dialog boxes from the common gray to a gentle but different hue—say, light yellow or light blue. Or, instead of changing the background to a color, change it to an image. You must be subtle here, but try taking your company's logo, rendering it in a single hue screened back 80% on a background of the same color screened back 90%. Place the gizmos directly on that image. The gentle shading won't intrude on the work at hand, but the pattern will be easily recognizable.

If you are more adventurous, you can try using the "ownerdraw" capability of push-buttons and slightly alter their shape. For example, add a tiny tab on the top, snip off one corner or round off all of the corners. You must be very careful not to affect the basic shape and shading of the button so that it doesn't change its affordance as a button, but the uniquely modified shape also whispers your company's name.

There is no reason that your dialogs can't say "you" instead of "them" and still be just as usable and consistent as though you copied the big guys verbatim. It will help the user navigate visually and will contribute positively to the branded look of your product.

Installation

Whether you install software off a CD-ROM, several 3 ½" floppies, a file server or the Internet, the process you go through is basically the same, consisting of two steps. First, the software must be copied or loaded onto your local hard disk. Second, the software generally requires some initial configuration so that both you and it can work smoothly.

The installation process is a necessary evil. There is no constructive need for an installation process—it doesn't help the user to achieve his goals; it doesn't help the program to perform its functions. In fact, most software development teams don't really give the installation process much thought at all. The effect is that most installation programs are written as afterthoughts. They are rarely designed and almost never designed well. Some clever vendors have developed a market selling installation-program-making tools. These have tended to institutionalize the drawbacks of bad installation procedures. The dreary sameness of most installation programs somehow lends an unwarranted credibility to them. But installation should be treated as an opportunity to excel. Instead of the normal, demoralizing test of user patience, installation can be a chance for your program to show off its good manners and consideration for the user.

Most software development managers are fooled by the unproductive nature of installation programs, so that they fail to see this as an opportunity. Installation programs are the first part of the program that the user sees, and they give the user his first impression of your product. If your installation program is a showcase of effective user interface design, the user will be in a fine frame of mind as he begins to use your program in earnest. Conversely, if your installation program is given the same afterthought-design that most of them get, your user

will be disinclined to tolerate anything less than instant perfection in the balance of your product.

Some modern installation programs have been given a visual once-over by a graphic artist, so they look pretty spiffy. But little consideration has been given to the interaction, and certainly not from a goal-directed point of view. The nature of installation programs remains one of blindly interrogating the user, forcing him to make uninformed decisions, and of the program making selfish assumptions about the way the computer is used.

What is wrong with installation

Possibly the biggest failure of most installation programs is their blind refusal to see that the user may wish to *un*install the program at some time. Very few installation programs also know how to uninstall themselves, so the option is rarely offered. The installation process often includes writing dozens or hundreds of command lines of code into various configuration files. It forces the Program Manager to create special windows and icons to accommodate it. It may add or modify files in the operating system's private directories, but it has no facilities for even remembering what it did, let alone a facility for reversing its actions. Complicating the problem is the difficulty—in many cases, the impossibility—of fully uninstalling a program.

The typical installation program is usually quite stupid in the way it does its job. For example, an installation program might create a new icon in the Program Manager, and then, when it is rerun after a failure, will create a new, redundant icon. The installation program assumes that it is flawless and that mistakes or misapprehensions will never happen. It assumes that it only needs to be run once, and no thought is ever given to the possibility that it might have to be run subsequently. The program doesn't even bother to look around and see its own spoor.

Klingon battle-cruiser mode

Installation programs usually behave in what I call **Klingon battle-cruiser mode**. Klingon battle-cruiser mode is characterized by the program behaving like a shoot-em-up arcade game. You grab the joystick and the program puts a diabolical Klingon battle-cruiser of a dialog box up on the screen. The dialog

demands that you make an arbitrary (and usually irreversible) decision about something of which you are completely ignorant. If you answer correctly, the installation program gives you ten thousand points, and another Klingon battle-cruiser dialog box appears. This time it asks you which interrupt vector is not conflicted with your SCSI adapter and, if you choose right, the dialog box disintegrates into space dust and you can proceed to the next Klingon battle-cruiser dialog box in the sequence. You never know how many Klingon battle-cruiser dialog boxes you are going to get, and you don't get much chance to go back and ponder. You never know when you might answer one incorrectly and the pesky Klingons will format your hard disk, scramble all of your INI files or just unceremoniously dump you at the C> prompt. All you see is a seemingly endless sequence of cryptic dialog boxes asking you for information you don't have and don't want to know.

Edward Tufte, author of *The Visual Display of Quantitative Information* (Graphics Press, 1983), detests the Klingon battle-cruiser mode, calling it "one damn thing after another." The program metaphorically pushes you into the hard-backed chair, aims the bare light bulb right into your eyes, menacingly slaps its palm with a rubber hose, and then proceeds to demand answers to difficult questions. This hyperbole may seem a little thick to those of you who work with computers on a daily basis (particularly you programmers out there), but this is really what it feels like to most users. It isn't pleasant. It isn't nice. It doesn't generate customer loyalty, and customer loyalty is what generates word-of-mouth sales and repeat buys and upgrades. In other words, it costs you money. You can always sell your product once if it does the job, but you won't last in this business if your product and your company don't generate customer loyalty. The industry is littered with companies that owned the market but failed to claim their customer's loyalty. I've already mentioned WordStar, née MicroPro, whose WordStar word processor dominated the market in the early '80s. As soon as WordPerfect and Microsoft Word became available, WordStar hit the skids faster than you can say Lotus Symphony. Digital Research's CP/M operating system had well over 90% of the microcomputer market in 1981. Within four years, the company was on the ropes and CP/M was just a fading memory.

Most, but by no means all, big manufacturers of shrink-wrapped software have at least attacked these problems in the last few years. Notably, the installation of Windows 95 shows a quantum leap forward in installation design.

The most common problems exhibited by installation programs are a microcosm of some of the nastiest software interface design problems in general. Most installation programs exhibit at least several of these design errors:

- Demanding responses without informing you of the consequences of your actions
- Not informing you of the scope of your actions
- Asking you questions about things to which you are unlikely to know the answer
- Asking you for answers that the program can determine for itself
- Acting really stupidly
- Not failing gracefully
- Not providing for uninstallation
- Ignoring evidence of their previous activity
- Abusing system-wide INI files
- Putting files where they don't belong
- Overwriting shared files
- Not offering you any information about the program
- Confusing installation with configuration
- Demanding your active participation

I'll now discuss each of these transgressions in detail.

Demanding responses without informing you of the consequences of your actions

Without a doubt, this is the most common of all of the transgressions of installation programs. It is the essence of Klingon battle-cruiser mode. The installation program puts up a dialog box that looks something like the one in Figure 32-2.

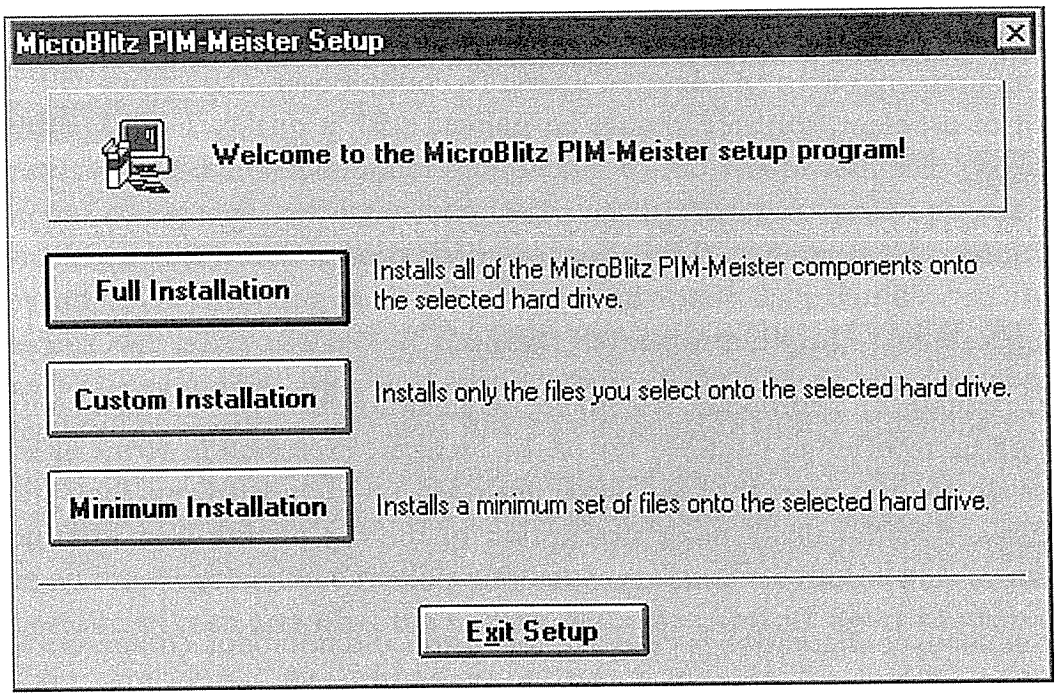


Figure 32-2

This is a typical installation program's first dialog box. Like playing a video game, you have only your wits to guide you. Is a full installation too much for me to handle? Am I smart enough to customize this program? Does it make me a wimp if I choose a minimum installation?

The program starts right off by asking you a question that will clearly have global consequences, is probably not reversible, and of which you have no understanding of the effects. Some more-advanced installation programs, notably those from Microsoft, make a pretty good disclosure of the effects of your choice on how much disk space will be consumed. However, you are still guessing about the meaning of the choice. What the user needs to know is exactly what functionality he will be sacrificing if he chooses a minimal installation. It isn't enough to merely know the disk space implications, he must know the usefulness implications of his choice, too.

Some versions of this question deal with system-level resources such as interrupts, communications ports, video drivers and the like. These are particularly vexing, because making the wrong choice can instantly lock up the computer system, crash other running programs, lose data, and sometimes even require rebooting the computer from a boot diskette and manually fixing the damage

done by the installation program. Although the consequences for making a wrong choice are severe, the user is rarely made aware that this is not the time for a guess—not even an educated guess. To counter user interactions with this kind of problem, software should be imbued with a quality I call **informed consent**. The user should only be asked questions about which he understands the consequences. In particular, he must understand the consequences to *him*, not just to his computer. If the program offers configuration choices, the user must be well-informed about how the various configurations affect the program's ability to help him achieve his goals.

For example, an appropriate way to create an atmosphere of informed consent would be to offer an itemized list of the features, expressed in terms of what they do for the user, that are either included or excluded from the various choices. Additionally, a prose description of the big picture from the user's point of view would be necessary. Something like the following would be nice:

"A minimal configuration is designed for laptop and notebook computers with available disk space of less than 100 megabytes. The MicroBlitz PIM-Meister will consume about 40% less space on your hard disk without sacrificing any critical functions. What you will sacrifice includes: most, but not all, online help text; the tutorial program for beginners; four out of seven Wizards; and most of the more obscure import and export utilities. If you want the minimal configuration but feel that you must have one or more of the excluded facilities, you can easily request the minimal configuration with *special options* and add the desired facilities back in. Also, you can always easily change your existing configuration by running the installation program a second time."

First, this statement describes the main reason why the user might want to choose the "minimum installation" option. Second, it describes in some detail exactly what is sacrificed to get it. Third, it informs the user how and why he can override the setting if he wants to. Fourth, it reassures the user by informing him how he can change things if he later changes his mind. This is informed consent, and the user will be able to make intelligent choices and feel good about them. Can you write the corresponding paragraphs for the other two installation choices?

Not informing you of the scope of your actions

A typical installation program wastes no time on what a programmer considers idle chit-chat with the user, but that chit-chat is important to dispel the user's

uneasiness. Imagine if an appliance repair person arrived at your house and, without a word to you, started wrenching apart your plumbing and dismantling your refrigerator. You would feel much better if the repair person gave you the big picture first:

"The compressor on your refrigerator is completely dead because the motor has seized. I will have to replace it completely. I have the replacement motor in my truck and it will take about an hour-and-a-half to make the repair, including recharging the system with coolant in an environmentally, friendly way. Your warranty will cover the cost of the parts but not my labor. I charge $45 per hour. The plumbing will need some minor work because the icemaker is directly connected to your pipes. Any time existing iron-pipe plumbing is disturbed, there is a slight risk of starting leaks elsewhere in the system. I will make every effort to keep the pipes from moving to reduce that chance."

You are now informed of how much time the operation will take; how much money it will cost; and what the risks of failure are. You are aware of the scope of the operation.

Wouldn't it be nice if our install program told us something like this:

"I am going to install MicroBlitz PIM-Meister on your system. This means copying the program from the distribution floppy diskettes onto your hard disk, decompressing the files and then configuring the program for your specific needs. I will need you to place the seven diskettes into the floppy drive one at a time as I read them. I will ask you for each one by the name shown on the diskette label. Judging from the speed of your processor, I estimate the entire process will take about 17 minutes for a full install and as little as 11 minutes for a minimal installation. The program will occupy between 8 and 14 megabytes on your hard disk, depending on which configuration you choose. In other words, it will take between 2.7 and 4.7 percent of your total capacity. Your disk is currently less than half full, so the available space will be reduced by 4.6 to 8 percent.

"I will place all parts of the program and all associated information in a special directory that I will create new. You will be given the choice of where that directory is located and what it will be called, but you can also just keep the default of PIMMEIST. By necessity, I must make at least one entry in your system files (in WIN.INI). This entry will be restricted to a single parameter line

that will have absolutely no effect on the system or any other program, even if you later decide to uninstall PIM-Meister. If you select the checkbox, I will also create an icon for launching PIM-Meister in the "Main" group of the Program Manager.

"This installation program maintains an internal status log so that, in the unlikely event it crashes, it will know how to pick up the pieces intelligently if you merely re-run it.

"You can uninstall this entire program at any time simply by pushing the uninstall button. The program will be removed from your system, leaving as few traces as possible, but leaving any data files you created with PIM-Meister untouched. If desired, you may request the space-saver uninstall, where the program is removed but all of your personal settings are saved. A subsequent execution of the install program can put PIM-Meister back exactly the way it was."

This monologue is pretty prolix, and experienced users won't want to read it. But new users will find it very reassuring to hear from the horse's mouth what the implications of the installation process are. They will understand the scope of the process they are about to undergo.

Asking you questions about things to which you are unlikely to know the answer

A typical installation program question for a communications program asks the user to specify the desired serial port. Most users don't know what a serial port is or how many they have, let alone which one is best for this program. Game and sound-card installation programs frequently ask users about available interrupt vectors, a question that can't be adequately answered by most computer engineers, let alone a typical home user. Business software installations can be expected to ask about network support and the type of mouse in use. Most users are completely unaware of the answers to such questions.

Asking the user a question to which he is unlikely to know the answer is very bad practice. First, it doesn't get the program the answer it needs. Instead, it gets a guess. Second, it makes the user feel bad—after all, he just failed a test and proved himself inadequate in front of a machine. How embarrassing! Third, it shows the program to be pretty stupid. It's like seeing somebody wandering on the street asking strangers "how many ergs are there to a joule?" What a dufus! Nobody knows the answer and nobody cares and there are better ways to find out stuff like that.*

*10^7

If the program needs to know about serial ports, it should test them and see which ones are occupied. It can search in various configuration files for clues as to how they are currently being used. It can even ask the user to move the mouse and look for activity on the various ports to eliminate that possibility.

If the program needs to know about interrupts, it can examine them or listen to them to determine whether they are in use or available. The installation program can make a very good guess (and probably a much more reliable guess than the user's) about what interrupts are available by deduction and by looking at system information files. By recording its findings and then telling the user it is about to do something that might lock up the system, the user can close all other running programs and be prepared for the program's error. If one occurs, he can then rerun the program, and it will find its earlier notes. It now knows what *didn't* work, which should be enough to enable it to deduce the correct choice. Don't force this choice onto a user who cannot be expected to know the answer.

Asking you for answers that the program can determine for itself

This is a common problem in all software, but the authors of installation programs are deservedly notorious for it. The program asks you what type of display device your computer has, when it can easily check inside Windows for the answer to that question. The program asks you how much disk space is available, when it can interrogate the file system to get the exact answer. The program asks you where another program or file is located, when it can easily search the disk to find it. The program asks you which hard disk you want to install on, when you only have one hard disk.

The computer's job is to remove unnecessary trivia from our lives. Questions like these only add more pointless trivia and are offensive. Most information needed by any program can and should be determined without asking the human user.

Acting really stupidly

I wish I didn't have to write this paragraph, but I do. There are many installation programs that begin the process of copying the program from the floppy

to the hard disk without first checking for a sufficient amount of free disk space on the destination drive. The program then blows up on a disk-full error. This is tantamount to walking into a post.

There are other ways that installation programs can get really stupid, like installing a Windows application on a system that doesn't have Windows or blindly configuring a program for color on a monochrome system.

One of the most obnoxious ways an installation program can behave is by not being aware of its own existence. The installation program goes ahead blithely installing an identical copy of a program that already exists. The installation program doesn't know that it is being used just to change a configuration, so it goes ahead and copies files from floppy to hard disk that are already there. The installation program doesn't realize that it blew up previously and is being re-run, so it mindlessly retraces its steps until it blows up again in the same way.

The designer of an installation program should write a list of all of the environmental givens that the program needs, including such things as RAM, video, disks, microphones, joysticks, mice, modems or speakers. The installation program should then check that these assumptions are indeed true before proceeding. The program should perform a commonsense examination of the system before it starts working. It should look for previous copies of itself; it should look for other, required software; it should check for fatal or dangerous conditions like lack of memory or disk storage.

Just yesterday, I installed a program (from a major vendor) that offered me the opportunity to put the program in either a default directory or one of my choosing. I chose one different from the default. Unfortunately, after I had installed the system, I found that significant portions of the program wouldn't work because the program would only look in the "default" directory. I had a hard time figuring out which was more stupid: That they couldn't find files on a hard disk or that they offered me an option that would cause fatal problems if I should choose to use it.

It isn't that hard for a program to find files on disk. If there is a name collision, the program can easily open the file and look inside to see if it is in the expected format. For any program to not be able to find its own files is just plain stupid.

Not failing gracefully

Because installation programs are often built as afterthoughts, they don't get the polishing, testing and refining that the main program gets. In particular,

installation programs have a penchant for crashing in very catastrophic ways. Where a regular application will report to the user that it is running out of memory before gently and safely divesting itself of unnecessary functions, features and data in order to continue, the average install program is very brittle. When it runs out of memory, it is not aware of the problem and merely dies, usually taking the rest of the system with it.

First, the memory requirements of an installation program are usually very static and predictable, so running out of memory is usually very avoidable. But if the program does run out, it should be sufficiently robust to detect the problem, make a permanent note of its current position so it can resume its task, then inform the user of the problem and give him the opportunity to adjust things and give the program another chance.

Most installation programs can only do an entire install process from scratch. If the installation consists of decompressing and copying the contents of ten floppies onto the hard disk and the program crashes on disk ten, it usually requires that the user pointlessly recopy the first nine floppies. The program should be smart enough to, before it begins copying, check to see if the copy process is really needed.

Not providing a means for uninstallation

Although many software vendors seem unaware of the fact, customers often want to remove software from their computers. A customer would remove software because it is needed on another computer; because it is not needed on this computer any longer; because it takes up space needed by other, more important, programs; and even because the user has decided that the program is not good enough to keep.

Every vendor should provide a tool for removing their program, just as they provide a tool for installing it. The uninstallation tool should be just as robust and full-featured as the installation tool. It should follow the principle of informed consent, telling the user what the scope and consequences of the program are.

The uninstallation program should remove all traces of the program where possible, including any entries made into system files such as WIN.INI and AUTOEXEC.BAT. If entries to these files are—or might be—shared with other programs, they must remain untouched. If the installation procedure created directories, and if no user-created files are in them, those directories and their

contents should be deleted. Those directories should be deleted from the PATH variable, too. If the program put anything in any directories outside of its own (a very bad practice), those items should be hunted down and deleted. The uninstallation process should also remove all files and directories created by the application program and not just those created by the installation program. Of course I don't mean any files created by the program for the user, like documents. Those—and their directories—must stay around for whatever plans the user has.

If the installation program loaded any dynamic link libraries (DLL) that are shared resources, it shouldn't delete them, unless it can absolutely confirm that no other programs depend on them. The uninstallation program should alert the user if it leaves DLLs behind but can't confirm whether they are used by other applications. If the user knows, he can then do it manually. In the current state of the system, it is usually impossible to know whether it is safe to delete a DLL, so this must remain just a goal for now.

It is reasonable to assume that the user does not hate your program, but is removing it because he merely wishes to regain some disk space by removing a no-longer-needed tutorial or some subsystems that have proven unnecessary. The uninstallation program should give the user the ability to remove individual pieces of your program without affecting the main function of the application.

The user may wish to move the application to a different hard disk or to a different place on the same hard disk. The uninstallation program should know how to make the transfer so that all references are updated and the program works smoothly despite the transition. Such references include the WIN.INI file references, including the file association for that extension, the GRP file references used by the Program Manager and the Registration Database.

If a user needed to temporarily reclaim the space on disk occupied by the program (say he needed additional free space for a two-week business trip), the uninstallation program should make this easy to do. It should remove all of the big, space-consuming files but leave behind all of the directories, configuration files and entries in other system files. When he returns from his business trip, the installation program can be used to put the big files back on his disk without overwriting or forgetting about his personal settings. The program would be reinstalled just as before.

Some programs such as networks, peripheral drivers and printer-sharing software are not only installed on the hard disk, but are also activated by the AUTOEXEC.BAT or CONFIG.SYS files at boot time, thus becoming a permanent part of the operating system. Software like this makes a special demand on its uninstallation program. It must be able to disable the program without physically removing it. If the user wants to, for example, run a game program that demands absolutely all available memory, the user should be able to disable the network drivers for the duration without physically removing them from disk.

Ignoring evidence of their previous activity

Installation programs should keep a log of their activity on the user's hard disk. This log tells the program what it has done before and what it is doing now. If the program learns anything from a previous execution, either by testing the system or by asking the user, it should be recorded here so reprocessing can be speeded up and the user doesn't have to be bothered again. The new installation facility for Windows 95 has this feature that Microsoft calls "SmartRecovery" and it works well. I hope it sets a standard for the entire industry.

Abusing system-wide INI files

Application software programs should limit themselves to no more than two or three lines of information in system-wide files such as WIN.INI, AUTOEXEC.BAT and CONFIG.SYS. If the program needs more information, it should create its own INI file and store the information there. In almost all cases, there is no need whatsoever to put anything at all in the system files and, if this is the case in your situation, please refrain from doing so.

Putting files where they don't belong

The application should operate in its own directory. If it requires multiple directories, they should be made subordinate to the program's main directory. The program should never put files in any other directories, particularly the WINDOWS directory or any of its subdirectories, the DOS directory, and the root directory. If the user were to install a new version of Windows or DOS, for example, the application's files might very well be deleted in the process. The resultant malfunctioning and confusion would be very unpleasant and completely avoidable.

Application programs often ignore the possibility that the operating system will be reinstalled or upgraded. The program is often inextricably dependent on entries in system files that, when the OS is reinstalled, will disappear. Most programs then require a complete reinstallation, including redundantly re-copying the files, when all it really needed to do was rewrite a line or two in the WIN.INI file. Better yet, the program should work without any entry in that—or any equivalent—file.

Overwriting shared files

Many applications use run-time libraries of some sort. Visual Basic applications in particular use the VBRUN dynamic link library (DLL) and usually a few VBX or OCX DLLs for each of the installable controls used in the program. When a program installs the DLL or VBX, it may overwrite one with the same name already installed by another program. For example, if program A uses a commercially available VBX grid control named "GRID.VBX" and program B, from another vendor, uses a proprietary VBX grid control that is also named "GRID.VBX," the installation process will cause problems. Even though the names are the same, the functionality and interfaces may be quite different. When program B is installed, it must ensure that it doesn't just overwrite the GRID.VBX file by assuming that it is an earlier version of itself. If it makes this assumption, program A will crash violently and mysteriously. The ensuing confusion will leave the user perplexed and angry.

A similar problem arises if two different programs use different versions of the same DLL. Imagine that both programs A and B use Version 1.0 of a DLL called DATBASE.DLL. The user then purchases the newest release of program B which includes the newest release of DATBASE.DLL, Version 2.0. The installation program for B likely assumes that it can blithely replace Version 1.0 of DATBASE.DLL with Version 2.0. But program A won't know how to deal with the new version of the library and a tragic crash is unavoidable. Crashes like this are particularly insidious, because the user could install the new release of program B in January and not get around to running program A until June. He will have no clue as to what caused the problem. If anything, he will blame the completely innocent program A. Because this problem can affect any vendor, even though they are not strictly at fault, it means that you must take defensive action to keep the problem from happening to you.

The problem can be avoided by following two simple guidelines. First, use names that aren't likely to collide with those from other vendors. Instead of naming a library "GRID," try using "XGRID," "GRD" or even "G7QL." The user will likely never see these names, so they don't have to be mnemonic. Second, append a unique number indicating the version of the library. Name the first release "XGRD1," the second release "XGRD2," and so on.

Just yesterday, this problem happened to Alice, our seminar manager. She installed a brand new copy of Adobe PhotoShop on her plain, vanilla IBM Aptiva computer, the computer crashed, and now she cannot get Windows to run at all. The installation program diddled evilly with something in the system—none of us in the office can figure out what—but there is no recovery path.

Not offering you any information about the program

The installation program should keep the user informed at all times of what it is doing and what remains to be done. Many contemporary installation programs put a dialog box on the screen with a small completion meter showing the amount done expressed as a percentage. Most of these meters are frustrating and confusing, however, because they don't explain what it is they are saying. Does the meter show the progress for this file? This section? This floppy? The copy process, but not the configuration process? The entire installation? Is it expressed as time or as bytes copied? Users have become used to being burned by meaningless meters, and they know enough to ignore them. They know that the installation program is just being stupid.

Confusing installation with configuration

Because most programs need some rudimentary configuration before they can run well, the installation procedure usually includes a configuration step. This is reasonable, but the configuration process is one that may need to be performed more than once, whereas the copying of files from the floppy is usually just a one-time thing. Installation designers frequently forget this and intertwine the two processes so that a reconfiguration cannot be done without an unnecessary re-copy operation. The installation program should be smart enough to recognize that it is being rerun and offer the user the option of just reconfiguring the existing instance of the program, without incurring all of the unnecessary overhead of re-copying.

Demanding your active participation

Some installation programs make unreasonable demands on your time and attention. They require that you actively participate in the installation process, even though you would just as soon delegate the job to the software. The WordPerfect for Windows installation program, for example, intersperses questions for the user with the actual copying of the program onto the hard disk. In other words, it asks you a question, then installs a few files, then asks you another question, then installs a few more files. The effect of this is incredibly annoying, because you are forced to consciously baby-sit the entire process. It should just ask you all of the questions at the beginning, then let you get up and walk away while the installation proceeds. If the install is from floppies instead of from CD-ROM or a file server, you still must stick around and feed it disks every few minutes, but at least you can let your mind roam, read or think about something else. Of course, it should also issue an audible alert every time a new floppy disk needs to be inserted, releasing you from having to watch the screen.

33

Shouldering the Burden

Because every instruction in every program must pass single-file through the CPU, we tend to optimize our code for this needle's eye. Programmers work hard to keep the number of instructions to a minimum, assuring snappy performance for the user. What we often forget, though, is that as soon as the CPU has hurriedly finished all of its work, it just waits, idling, doing nothing, until the user issues another command. We invest enormous efforts in reducing the computer's reaction time, but we invest little or no effort in putting it to work proactively when it is not busy reacting to the user. Our software commands the CPU as though it were in the army, alternately telling it to hurry up and wait. The hurry up part is great, but the waiting has got to stop.

The division of labor in the computer age is very clear: The computer does the work and the user does the thinking. Computer scientists have focused our attention on artificial intelligence, tantalizing us with visions of computers that

think for themselves. This pursuit is a rewarding intellectual exercise for computer scientists, but users don't really need much help in the thinking department. They do, however, need a lot of help with the work of information management, activities like finding and organizing information, but the actual decisions made from that information can best be made by the wetware—real people.

The computer does the work and the user does the thinking

There is some confusion about "smart" software. Some naive observers think that smart software is actually capable of behaving intelligently, but what the term really means is that these programs are capable of working hard even when conditions are difficult. Essentially, smart software is a lot less brittle than other software. When we create smart bombs, after all, we create bombs that will hit *our* targets with great precision under trying circumstances, not bombs that make their own decisions about which target to hit.

The opportunity is in work, not in thinking

Regardless of our dreams of thinking computers, there is a much greater and more immediate opportunity simply in getting our computers to work harder. I'd like to see us give our software the virtue of conscientiousness. A conscientious person has a larger idea of what it means to perform a task. Instead of just washing the dishes, for example, a conscientious person also wipes down the counters and empties the trash because those tasks are also related to the overall goal: cleaning up the kitchen. Instead of just drafting a report, a conscientious person puts a handsome cover page on it and makes enough photocopies for the entire department.

Most of our existing software, regardless of the power it can bring to bear on a given task, is not conscientious. It has a very narrow understanding of the scope of most problems. It may willingly perform difficult work, but only when given the precise command at precisely the correct time. If, for example, you ask the inventory query system to tell you how many widgets are in stock, it will dutifully ask the database and report the number as of the time you ask. But what

if, twenty minutes later, someone in the Dallas office cleans out the entire stock of widgets. You are now operating under a potentially embarrassing misconception, while your computer sits there, smugly idling away billions of wasted instructions. It is not being conscientious. It doesn't have to be "intelligent," artificially or otherwise, to help you out here. If you want to know about widgets once, isn't that a good clue that you probably will want to know about widgets again? You may not want to hear widget status reports every day for the rest of forever, but maybe you'll want to get them for the rest of the week. It doesn't take a neural network to lend a helping hand.

In our current computing systems, the user has to remember the names he gives to files and where he puts them. If he wants to find that spreadsheet with the quarterly projections on it again, he must either remember its name or go browsing. Meanwhile, the processor is sitting there, wasting billions of cycles, without even bothering to lift a register to help. Software just doesn't seem to want to work very hard or help out. When the user is struggling with a particularly difficult spreadsheet on a tight deadline, for example, the program offers precisely as much help as it offers when he is just noodling with numbers in his spare time. All of the user's human colleagues know that the job is critical, so they help if they can or at least stay out of his way. But not the program. It just plods along, oblivious to the storm brewing over its head.

This model has to stop. There is a lot of work to be done, and the software can no longer in good conscience spend so much time twiddling its digits while the user works. It is time for our computers to begin to shoulder more of the burden of work in our day-to-day activities.

Let's put those idle cycles to work

Software is designed to perform sequentially, one action after another, and as each chunk of code talks to other chunks of code, they wait for the response. This is fine for code, because the wait is a few millionths of a second. If there's a peripheral involved, the wait might be a few thousandths of a second, but it's still quite fast. However, if the program tosses the ball into the user's court, things are dramatically different.

Most normal users in normal situations can't do diddlysquat in less than a few seconds. That is enough time for a typical computer to execute a few dozen *million* instructions. Almost without fail, those interim cycles are dedicated to idling. The processor does nothing except wait. The argument against putting

those cycles to work has always gone something like this: "We can't make assumptions; those assumptions might be wrong." Our computers today are so powerful that, although the argument is still true, it is frequently irrelevant. Simply put, it doesn't matter if the program's assumptions are wrong, it has enough spare power to make several assumptions, and merely toss the results of the bad ones out as soon as the user makes his choice.

Now with Windows 95's preemptive, threaded multi-tasking, you can perform extra work in the background without affecting the performance the user sees. The program can launch a search for a file, and if the user begins typing, merely abandon it until the next hiatus. Eventually, the user stops to think and the program will have time to scan the whole disk. The user won't even notice.

Every time the program puts up a dialog box, it goes into an idle waiting state, doing no work while the user struggles with the dialog. This should never happen. It would not be hard for the dialog box to hunt around and find ways to help. What did the user do last time? Maybe the program could offer the previous choice as a suggestion for this time. Maybe the program could count and display the number of occurrences of widgets in the program. Maybe it could report on the history of the widget. Maybe it could find out if other users had accessed the widgets. If it stayed up for several seconds without activity, it could offer more assistance.

Memory

In Chapter 14, I talked at length about giving your program a memory. This is the primary tool for making your program shoulder more of the burden of work. Everything that happens should be remembered. There is plenty of storage on our big hard disk drives, and a memory for your program is a good investment of storage space. We tend to think that programs are wasteful of disk space, because a big horizontal application might consume 30 or 40 MB of space. That is typical usage for a program, but not for user data. If your word processor saved off 1 KB of execution notes every time you ran it, it still wouldn't amount to much. Let's say that you use your word processor ten times every business day. There are approximately 200 work days per year, so you run the program 2,000 times a year. The net consumption is still only 2 MB, and that gives an exhaustive recounting of the entire year! Most screen savers take ten times that much storage!

All file-open facilities should remember where the user gets his files. Most users only access files from a few directories for each given program. The program should remember these source directories and offer them on a combobox on the file-open dialog. The user should never have to step through the tree to a given directory more than once.

And don't just remember the explicit things, also remember what can be deduced from explicit things. If the program remembers the number of bytes changed in the file each time it is opened, it can help the user with some reasonableness checks. Imagine that the changed-byte-count for a file was 126, 94, 43, 74, 81, 70, 110, 92. If the user calls up the file and changes 100 bytes, nothing would be out of the ordinary. But imagine if the number of changed bytes suddenly shoots up to 5,000. The program might suspect that something was amiss. Although there is a significant chance that the user has inadvertently done something about which he will be sorry when compared to the other edits, the actual chance of a problem is low, so it isn't right to bother him with a confirmation dialog. It is, however, very reasonable for the program to make sure to keep a milestone copy of the file before the 5,000 bytes were changed, just in case. The program probably wouldn't need to keep it beyond the next time the user accessed that file, because the user would likely spot any mistake that glaring the next time he accessed the file and would demand an undo.

For that matter, most programs discard their stack of undo actions when the user closes the document or the program. This is short-sighted on the program's part. Instead, the program could write the undo stack to a file. When the user reopens the file, the program could reload its undo stack with the actions the user performed the last time the program was run—even if it were over a week ago. These actions are likely not needed, but in the rare case where they are, it's certainly better to have them than not to, and the processor was just idling anyway.

Stamp out foolish software behavior

I want to tell you about my imaginary secretary, Rodney. If I hand Rodney a manila folder and tell him to file it away, he checks the writing on the folder's tab—let's say it reads "MicroBlitz Contract"—and proceeds to find the correct place in the filing cabinet for it. Under M, he finds, to his surprise, that there is a manila folder already there with the identical "MicroBlitz Contract" legend. Rodney notices the discrepancy and investigates. He finds that the

already-filed folder contains a contract for 17 Doppelgängers that were delivered to MicroBlitz four months ago. The new folder, on the other hand, is for 32 Relativity-Podensers slated for production and delivery in the next quarter. Conscientious Rodney changes the name on the old folder to read "MicroBlitz Doppelgänger Contract, 7/97" and then changes the name of the new folder to read "MicroBlitz Relativity-Podenser Contract, 11/97." This type of initiative is why I like the conscientious Rodney.

My former imaginary secretary, Elliot, was a complete idiot. He was not conscientious at all, and if he were placed in this same situation he would have just dumped the new "MicroBlitz Contract" folder into the cabinet next to the old "MicroBlitz Contract" without a second thought. Sure, he got it filed safely away, but things could have been done better. That's why Elliot isn't my imaginary secretary anymore.

If, on the other hand, I rely on software, my word processor, to draft the new Relativity-Podenser contract and then try to save it away in the MicroBlitz directory, the program offers me a Hobson's choice of overwriting and destroying the old Doppelgänger contract or just not saving it at all. The program not only isn't as capable as Rodney, it isn't even as capable as Elliot. It is stupider than a complete idiot. Even Elliot didn't come unglued when he found the old, identically named folder. Even Elliot figured out how to make do on his own initiative (small though it might have been) without having to bother me with silly demands and pointless details. Only the software is dumb enough to make the assumption that because they have the same name, I meant to throw the old one away. The choices the software offers are terrible! There is simply no excuse for behavior this dumb.

The program should have, at the very least, taken the same action that Elliot did, and merely marked the two files with different dates and saved them away. Even if the program refuses to take this "drastic" action unilaterally, it could at least show me the old file, letting me rename *that* one before saving the new one. There are numerous actions that the program can take that aren't as insultingly stupid as what it currently does. They are easy to think of, too. All you have to do is pretend the software is human.

Things your program can do in the background

Each application could leave a small thread of itself running between invocations. This little program could keep an eye on the files it worked on. It could

track where they go and who reads and writes to them. This information might be helpful to the user when he next runs the application. When he tries to open a particular file, the program can help him find it, even if it was moved. The program could keep the user informed about what other functions were performed on his file, like whether or not it was printed or faxed to someone. Sure, this information might not be needed, but the computer can easily spare the time, and it's only bits that have to be thrown away, after all.

Microsoft's new Plug-and-Play standard is a move in this direction, because it demands that programs be cognizant of the changing world around them. If the user plugs in a sound card or a tape drive, programs should notice the difference immediately.

If the user resizes a pluralized application, the program can take the time to quickly rearrange gizmos on the toolbar and items on the menu so that they make better use of the new, more-restricted space. Even better, if the program keeps track of the frequency of each gizmo's use, the ones very rarely used can be omitted to make more space for the more needed ones.

Galleries

Many programs offer tools to users. That is fine as far as it goes, but why can't the program offer complete solutions, too? A program that lets you configure your own personalized newspaper from information on the Internet is nice. Some users will really appreciate being able to put sports at the top of page one. Most users, however, will probably want a more traditional view, with world news at the top and sports at the back. Even these more-traditional users will appreciate the configurability, though, so they can add their local news and news concerning topics of particular personal interest. Typically, though, a configurable program offers the user a blank slate and a suite of tools for filling it. Our sports enthusiast won't see this as a drawback, since she planned on making extensive changes anyway, but most other users will. If they want something standard but with a few differences, they shouldn't have to start from scratch to get it. They should be able to pick a pre-made newspaper and then make the few small changes to it needed to get their custom version. The user should be allowed to choose a starting design from a gallery of likely designs.

Design tip: Offer the user a gallery of good solutions.

Some programs already offer galleries of pre-designed work, but many more should do so. Most users are intimidated by blank slates, and they shouldn't have to deal with one if they don't want to. A gallery of good, basic designs is a fine solution.

Natural language output

There are some functions found in software that are really difficult for users to control. For example, querying a database is always tough, because it calls for Boolean algebra. I talked about the problems with Boolean notation in Chapter 3. Just because the program needs Boolean queries, users shouldn't have to use it, too.

An alternative is to use natural language processing, where the user can key in his request in English. The big problem with this method is that it is not possible for today's run-of-the-mill computers to effectively understand natural language queries. It might work in the laboratory under tightly controlled circumstances, but not in the real world, where it is subject to whim, dialect, colloquialism and misunderstanding.

One approach that I have used successfully is what I call **natural language output**. Using this technique, the program proffers to the user an array of bounded gizmos to choose from. The gizmos line up so that they can be read as an English sentence. Notice, in this case, the user is merely choosing from valid alternatives and only the output is "natural." Figure 33-1 shows how it works.

Remember, English isn't Boolean, so the English clauses aren't joined with AND and OR, but rather with English phrases like "all of the following apply" or "not all of the following apply." The user finds that making the choices is easy because they are very clear and bounded, and when he is done, he can read it like a sentence to check its validity.

Extraction gizmos

In Chapter 27 I introduced the concept of extraction gizmos—text gizmos that are smart enough to recognize their contents. They are very consistent with the idea of having computers shoulder the burden, because they are willing to do more than just pass ASCII characters between the user and the program. They are willing to put some effort into breaking those strings of characters into meaningful pieces.

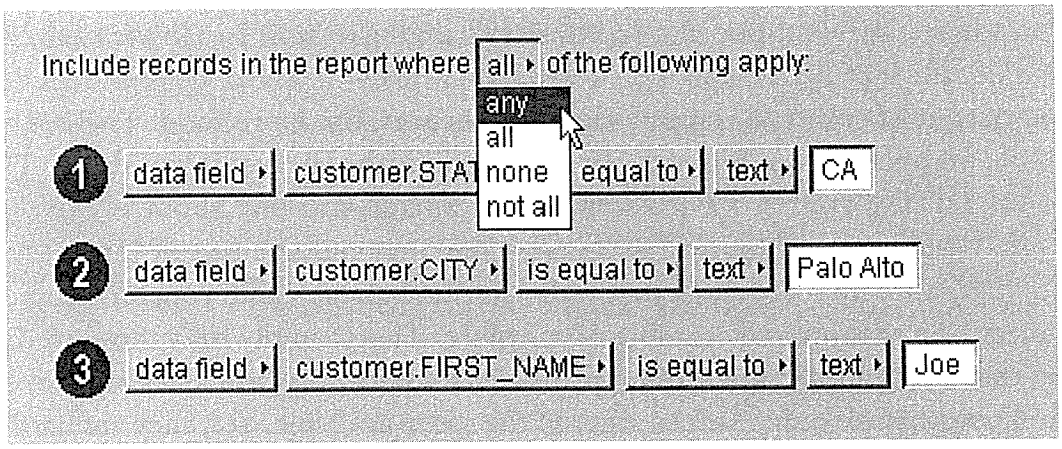


Figure 33-1

Here is a sketch of a dialog box that produces natural language as output, rather than attempting to accept natural language as input. Each button is actually a drop-down with a list of selectable options. Essentially, the user constructs a sentence from a dynamic series of choices that always guarantees a valid result. The user can enter text or numbers into the white fields.

There are dozens of types of extraction gizmos that could be written to handle dates, distances, currencies and similar things. There are probably several that would be uniquely beneficial just for your company's software; they could tackle input like employee numbers, job titles, part numbers or divisions.

See if you can design algorithms for processing input for other types of extraction fields. It seems daunting at first, but it is actually quite easy if you remember the simple fact that you don't have to decipher 100% of the information. It is okay to be 80 or 90% accurate. If the user enters information that is indecipherable, it means, by definition, that you can't decipher it. The user certainly cannot blame the software in this case. On the other hand, if the software doesn't even try, the user can, will and should blame the software.

Visual richness

Sid Meier is the genius designer at MicroProse who created the best-selling games *Railroad Tycoon* and *Civilization*, among others. I love his games; *Civilization*, in particular, is addictive. I strongly recommend that every software designer spend time learning from Meier's work.

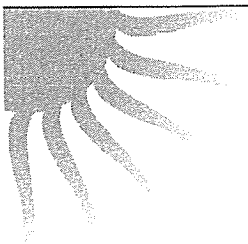
Sid Meier's programs are instantly recognizable because of the visual richness he gives them. Visual richness in games is no big thing in these days of

CD-ROMs and lush graphics, but Meier creates this richness with dynamic—rather than static—output that represents changing values. In *Civilization*, for example, he uses little icons that resemble sheaves of wheat to represent the food output of your civilization; the more sheaves, the more food. There is no specific number shown, but you can always count the sheaves if you'd like. The sheaves are displayed in a horizontal line in a fixed-size box about two inches across. When your civilization is young, there may only be five or ten sheaves in a row, and they are easy to count. As time passes and your civilization matures, the sheaves may number 50 or more. There simply isn't room to display that many in the space allotted. Meier's solution is brilliantly simple: he merely overlaps one sheaf on another in the space available. Yes, they are harder to count when they overlap, but when they overlap, who needs to count? He puts a short gap in the line of sheaves between the amount of food that is necessary for survival and the amount of food that goes towards future growth. With a glance, the length and density of the line of sheaves tells you how well your civilization is doing, and the relative position of the gap (or its absence) tells you your rate of expansion.

The sheaves are just one of the measures. Next to them are arrows for trade, shields for productivity, light bulbs for intellectual achievement, and more. Meier crams an enormous amount of quantitative information into a very small space completely symbolically. Any beginner can intuit that more symbols are better, without needing to understand the details to successfully play the game. However, if you find that you like the game (and who wouldn't?), you will play it more and more, and will inevitably find yourself wondering about those little lines of symbols. A quick check of the manual points out the secret of the gap and how they are counted, and you will immediately become a much better player.

Another of Sid Meier's masterpieces is *Railroad Tycoon*. In this game, you create a commercial railroad empire—setting up routes and running trains to earn revenue. The game has pleasant sound effects, with locomotive whistles blowing and bells ringing. After playing the game for many hours, I discovered that the whistles and bells weren't random, but that one sounded every time a train passed through a station. After more hours, I became curious about why there would sometimes be a bell and sometimes be a whistle. A quick check of the documentation (commensurate effort made me very motivated to read the manual) explained that a bell rings if the train makes money at that station and the whistle blows if there is no revenue. Wow! Immediately, I could tell the

financial health of my railroad just by listening. If the bell rang constantly with just a few toots, I was doing okay, but if there was a lot of whistle blowing with just a few dings, I could tell that I needed to reassign my trains and alter their cargoes.



Hot, simple and deep

Game publisher Trip Hawkins used to say that good games are hot, simple and deep. They are hot if they are interactive and fun. They are simple if they are easy to understand and use right away. And they are deep if they offer continual learning and challenge to the persistent user. This is a fine axiom for all software, not just games. The same way that Sid Meier densely crams dynamic information into the interface of his games, you can cram useful information into yours. You're not going to have sheaves of wheat, but you will have lines of little symbols indicating the number of cells your spreadsheet refers to in other sheets, for example. Or the number of records in the database could be indicated with tiny little dots. The number of updates to the current record could be shown with a row of tiny icons. None of these values are critical for the user's success, but as the users become more experienced with the program, they will appreciate the depth of its feedback. Don't disappoint them.

Get our software talking to our hardware

Some process dialogs just tease the poor user with their CANCEL buttons. In a typical print operation, for example, the program begins sending the 20 pages of a report to the printer and simultaneously puts up a print process dialog box with a CANCEL button. If the user quickly realizes that he forgot to make an important change, he presses the CANCEL button just as the first page emerges from the printer. The program immediately cancels the print operation. But unbeknownst to the user, while the printer was beginning to work on page 1, the computer has already sent 15 pages into the printer's buffer. The program cancels the last five pages, but the printer doesn't know anything about the cancellation; it just knows that it was sent 15 pages, so it goes ahead and prints

them. Meanwhile, the program smugly tells the user that the function was canceled. The program lies, as the user can plainly see.

The user isn't very sympathetic to the communication problems between the application and the printer. He doesn't care that the communications are one-way. All he knows is that he decided not to print the document before the first page appeared in the printer's output basket, pressed the CANCEL button, and then the stupid program continued printing for 15 pages even though he acted in plenty of time to stop it, and it even acknowledged his CANCEL command. As he throws the 15 wasted sheets of paper in the trash, he growls at the stupid program.

Imagine what his experience would be if the application could communicate with the print driver and the print driver could communicate with the printer. If the software were smart enough, the print job could easily have been abandoned before the second sheet of paper was wasted. The printer certainly has a cancel function—it's just that the software is too indolent to use it, because its programmers were too indolent to make the connection.

Too much software takes the attitude that "it isn't my responsibility." When it passes a job along to some hardware device, it washes its hands of the action, leaving the stupid hardware to finish up. Any user can see that the software isn't being conscientious, that the software isn't shouldering its part of the burden for helping the user become more effective.

34

Where Do We Go from Here?

Software marketing consultant Seymour Merrin says, "We found it easier to convince people that software was easy to use than it was to actually make it easy to use." There is sad wisdom in this observation. The power inherent in computers is forcing them into every industry and practice whether they are easy to use or not. It is up to us as users to demand better. The economics will force us to use them, but only acting on our conscience will make it pleasant and really effective.

Software sucks

In general, the software we use here in my office frustrates us intensely. We don't have a complex setup here, just a half-dozen relatively new computers running Windows 95 or Windows for Workgroups 3.11 on some vanilla network. Right now, Wayne is meandering the halls muttering to himself because the server is down again. Poor Wayne is really a Macintosh guy at heart, but he's reduced to specifying obscure text commands to the format program in

trying to resurrect the server's hard disk. Geetha, down the hall, is trying to learn Canvas, a drawing program. She's a user interface designer, and finds Canvas personally and professionally insulting because its interface is so horrendously bad. She would rather use PhotoShop, except that it doesn't let her draw good screen images for several reasons, mostly because it's designed for photo manipulation and not for screens with buttons and gizmos. I've got a problem on my computer, too, where the neat new power-saving features of the software are in conflict with the neat new power-saving features of the hardware. The net effect is that every few minutes my computer goes completely stupid for a few seconds while the disk wakes up again. The other result is that I can't turn the screen saver off, but when the screen saver runs, it crashes the system completely. If I leave my computer for more than an hour, it will die. Just a few minutes ago, my seven-and-a-half-year-old son Marty telephoned me in tears from home. He is trying to make his new Kid Cad drawing program work, but he can't make all of the "windows go away." The File Manager and the Program Manager are making his life miserable with meaningless excise. I'm not making this up. It's just a typical day with computers.

Not all of these are user interface problems, but all of them are problems that users must grapple with and solve. Frankly, I don't understand how non-computer-professional people can make computers work. This situation is really not acceptable. It's certainly not acceptable in the long run for the computer industry. The vendor who can solve the problem will surely win customers. Until then, bad software is our own fault, because we buy it, even though it makes fools of us.

Saved by the Net. Not.

People who should know better are getting excited about the potential of the Internet, the World Wide Web and Interactive Television to change the user interface landscape. Why should things be any different just because there are some underlying hardware and bandwidth changes? I am very enthusiastic about the potential of the Web, but I don't see much progress on the user interface front reflected there. I see the same old problems of stupid, rude, inappropriate software that hardly lifts a finger for the user. It just happens to be wired and have a high cool-quotient.

Ultimately, we will make good software by examining and satisfying the user's goals. We will not make it by moving to new platforms or by improving the

technology. Our technology is superb. What it lacks is some consideration of the human.

We know a lot about old technology

We now have more than a decade of refinements to the PARC paradigm. We know how to create good error messages, confirmation dialog boxes and buttcons. We don't have anywhere near as much experience in creating rich, visual, unified interfaces that work hard to support users. We have years of experience building systems with robust data integrity and sophisticated hierarchical file systems, but we don't yet have experience creating systems with data immunity and associative storage systems.

The problem is quite simple: Everything we know about computers is wrong! Forty years ago, there was less computing power on the entire planet than is in your wristwatch today. There is literally more computing power in your family car than there is in the space shuttle. Just twenty years ago, computers were precious commodities that were extremely expensive, limited and weak. In 1974, when I began working with computers, I cut my teeth on an IBM 370/135 mainframe that was absolutely brand new and state of the art. It had 144 KB of main memory. Yes, KB! It had two 100 MB hard disk drives, each the size of a big refrigerator. It had a card reader and a card punch and a chain printer. It resided in its own room, nestled deep inside its own building. The room had mostly glass walls, a raised floor, powerful air conditioning, three full-time operators and an IBM systems engineer who came around every two weeks to perform preventive maintenance. I learned the hard way that computing resources were always very scarce. In fact, all of the senior programmers and computer scientists in business or academia today learned this attitude, which I call **scarcity thinking**.

We all knew, deep in the fabric of our thinking, that there was never enough memory, never enough storage, never enough cycles and never enough bandwidth. We all wanted to be good at what we did, so we worked hard to maximize the scarcest resource: We made sure that the CPU got all of the breaks. We developed systems to maximize the use of disks, of RAM, even of punched cards. If you are too young today to be of this group, doubtlessly you were taught by this group, and the senior developers at your shop are probably members of this group, and most of the software you use was designed by this group. The men and women who know, deep down in their guts, that

computer resources are scarce are running the show and setting the pace in the software industry today.

My mother and father grew up during the Great Depression. To them, a steady, well-paying job was a luxury, and they had learned this lesson the hard way. They could never understand my entrepreneurial tendencies and my disdain for traditional employment. It was as though I disdained oxygen. When you have lived with real scarcity, you can never unlearn the lessons; they bury you with their scars.

Today, by comparison, our computing power is an embarrassment of riches. My little, aging desktop computer has 16 megabytes of main memory, 1.7 gigabytes of hard disk and a processor that can execute 66 million instructions per second. It has an order of magnitude more memory on its *video card* than that old 370 had in its main memory. Within a couple of years the state-of-the-market computer will have a processor ten times faster with ten times more memory and storage than even today's fastest and biggest. It will be connected to every other computer in the world by digital phone lines that pass data at tens of thousands of bits per second. We have, within a short score of years, left behind a world of scarcity and entered a world of abundance, with even greater realms of abundance just beyond the horizon. Our computers are as powerful as we want them to be. We have all of the bits and bytes and cycles we need to design software that really serves humans.

The opposite of scarcity thinking is **abundance thinking**, and good software designers will have this sense in their minds instead. Abundance thinking frees designers from worrying about memory, storage or cycles. They must worry instead about users, and they have the design sense and training to provide those users with interfaces and features that make them more effective.

The trouble is that we have constructed the entire industry in the image of that old, obsolete scarcity thinking. At the top of the heap are programmers who can work close to the metal; who can create software that maximizes the performance of the scarce hardware. At the bottom of the pyramid are unskilled users who haven't become "computer-literate." We sweep these people under the rug because they aren't as important as smoothing the way for the precious, struggling CPU.

The very fabric of our thinking is strongly and powerfully colored by this scarcity thinking. All of the software technologies we prize so much are really tools for relaxing the demands on memory, on storage, on processor cycles. This includes databases, networks and even the way we write files. For example, most human beings who handle paper forms file away a filled-out form, yet almost every computer program that files away filled-out forms first eliminates the form. It just files away the answers. This storage technique—used by virtually every program ever written—makes it easier for the disk drive. The fact that it simultaneously reduces the software's capacity to adjust, to adapt, to conform to the idiosyncratic behavior of humans, is considered merely a necessary cost of doing business. In a world of computer scarcity, this is a reasonable compromise with users. We ask users to tolerate the compromise because they are strong and computers are weak. But we don't live in this world of scarcity any longer. We live in a world of abundant computing resources, and computers are strong and users are weak.

Don't ask programmers to design while they code

Programmers are squeezed into a conflict of interest between serving the CPU and serving the user. They cannot successfully be asked to design for users because good coding demands that the CPU be serviced with a single-minded commitment. Inevitably, they will make judgments based on the difficulty of coding and not on the user's real needs. This is not to say that any given programmer can't make the right choice, just that programmer's can't make the right choice while they are actively employed producing the code that results from their choice. These tasks must be separated for the user design part to have a chance. Some of the greatest design ideas have come from programmers, but this is a happy accident. If we want good design industry-wide, we will not get it by accident.

A great basketball player can either play or be a referee, but he cannot be both simultaneously. My doctor has a doctor; he doesn't self-diagnose. The NFL wisely doesn't let players bet on games. Venture capitalists can't invest their personal money in companies. Politicians must put their investments in blind trusts. Judges cannot adjudicate cases for their friends and neighbors. We recognize the ethical quandaries created by a conflict of interest in most of our other activities. We just seem to ignore them in software development.

Even when the designer creates a solution, the programmer might go away and edit it independently. Anyone who has worked for a while with programmers has had this experience: the team meets and everyone agrees on the course of action. Everyone acknowledges their tasks and what the program will look like. Two weeks later, when the group reconvenes, a programmer says—without any trace of irony—"yeah, I decided to do it this way instead. I thought it was better," while the rest of the team gnashes their collective teeth. Even if it *is* better, it is still wrong to change things unilaterally when a team is depending on you. If a marketing-communications junior executive, for example, came back after two weeks with the same outrageous claim about a brochure, he would likely be fired. Generally, technical managers are protective of their programmers and refuse to call them on their willful behavior.

Solving the problem

One area where the software industry, including Microsoft, is making headway is with **usability**. Usability is a science, consisting mostly of empirical testing and observation of users interacting with software or prototypes of software. Those observations illustrate problem areas, which can then be addressed. Usability testing has been adopted by many companies in the last few years.

The chief drawback of usability is that it sidesteps actual design. The process of testing is very different from the process of design. Design springs directly from the knowledge of goals. Usability derives from specific objects. Usability testers refine what programmers create, rather than fabricating solutions from first principles. There is merit to this refinement—if Microsoft's recent efforts are any evidence—but limited post facto influence cannot change software down to its roots. And good user interface begins way down deep, not on the surface.

The other drawback of usability testing is that it leaves the programmers in charge. If you want to create a beautifully cut diamond, you cannot begin with a lump of coal. No amount of chipping, chiseling and sanding will turn that coal into a diamond. At best, the gentle sanding and polishing that results from usability testing will only give you an attractively carved lump of coal. Programmers have had unchallenged say over the software for too long, and usability testing leaves untouched the assumption that programmers should devise the point of departure.

One of the central tenets of usability engineering is that design should be "user-centered." This certainly sounds good, but it has serious problems. The

biggest problem is that it is widely interpreted to mean that your users can tell you how to design software. Saying "user-centered software design" is like saying "fish-centered aquarium design." You wouldn't ask the fish, would you? Although most usability professionals understand the distinction and don't let themselves get jerked back and forth by the results of focus groups and user tests, many others believe that what users say is gospel. What users say is generally goofy. They can only give faint indications of places where problems may exist. They do not have the training necessary to actually solve the problems.

Nathaniel Borenstein, in his book *Programming as if People Mattered* (Princeton University Press, 1991), says, "Listen to your users, but ignore what they say." This is a very accurate instruction for software designers. Users are filled with raw information. What they lack is wisdom, a sense of design, an understanding of the medium, a willingness to break out of the box of existing solutions, a language to express their desires and any experience in delivering software solutions. Borenstein goes on to say about those who ignore his advice, "The world is full, accordingly, of bad user interfaces that were essentially designed (or redesigned) by nontechnical people with no real idea of what they were doing, and implemented uncritically by programmers who were, like Adolf Eichmann scheduling deportations to the Nazi death camps, 'just following orders.'" I really admire his gutsy willingness to state this in such extreme terms. The key to good user interface design is not users, but user interface designers.

Anyone who has argued with a programmer knows how difficult it is. They are very intelligent and are only swayed by logic and reason. Unfortunately, you can't defend the user's needs very well with mere logic and reason. They aren't bad tools, just inappropriate ones for the problem at hand. Programmers will always design logically and rationally, and they will rarely produce good design. After all, those logical tools got us into this predicament.

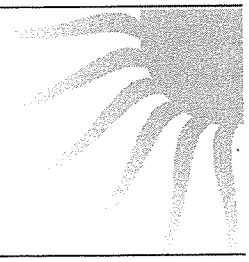
More than a few of the usability professionals I have met resist the idea of software design because it is done without much user testing. I suspect that these people are too used to fighting with recalcitrant programmers. Programmers fight desperately against the insistence that their creations can't be valid until they are tested by users, and usability professionals seem to have retreated to the empirical as their only way to convince the logical, rational, engineering mind that there is another, better approach to designing the user interface.

They drag programmers into dark rooms, where they watch through one-way mirrors as hapless users struggle with their software. At first, the programmers suspect that the test subject has brain damage. They cannot believe that any user could be so stupid as to not understand their program. Finally, after much painful observation, the programmers are forced to bow to empirical evidence. They admit that their interface design needs work, and they vow to fix it.

But empiricism is not a method of design, it is a method of *verification* of design. It is one thing to use blind testing as an indicator of problems. It is quite another to use it as a source of solutions. Forcing programmers to watch users struggle is good therapy for the programmer, but it doesn't do a heckuva lot for the user, or for the software. The programmers go right back to their computers and apply a bit more logic and reason to the user interface. We will only get significant quantities of well-designed software when its design is in the hands of software designers and not programmers; not even programmers assisted by user testing.

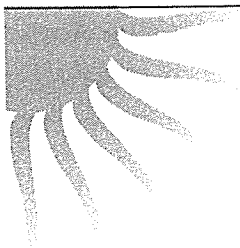
I have seen usability professionals run rigorous user tests, tests that were conducted with superb methodology. Upon reviewing the results, they said things like "Well, I'd prefer to see those gizmos lined up better," or "I guess we could move this button over here." User interface design is not guesswork. When good user interface designers create a dialog box, they know why. When they populate it with gizmos, they understand the purpose that each of them serves. And just as important, they know when to *not* create a dialog box.

User interface design is not guesswork



Most programmers design in one of two ways: they make guesses based on their programming expertise, or they copy from existing programs. Either way, they usually end up rendering the implementation model and trapping the user in a prison of technology. Usability testing responds to this guesswork with empirical observation. This testing ends up being guesswork, too, because the insights tend to come from users and not from designers. User testing is to user interface design what market research is to sales. You can dispense with one but

not the other, and market research can never substitute for sales. Likewise, user testing can never substitute for user interface design.



User testing can never substitute for user interface design

User testing can tell an observer when a program is too complex, too misleading or too confusing. Armed with this information, it is possible for a programmer to reduce its complexity and confusion, but it takes a designer to synthesize a proper solution from scratch. That solution will conform to the user's natural mental model, giving him what he needs in terms he understands. Designers have an understanding of how users think and also know how software is built. They bridge the two worlds with their natural design sense. In the same way that a programmer is born with a sense for how to imagine complex procedural systems, or an artist is born with a sense for creative expression, a designer is born with the vision to see what technology can offer in human terms.

This is a power struggle. Programmers have dominated all aspects of software development for decades, and for the first time they find themselves up against a problem—designing for users—that their trusty logical tools are failing to vanquish. Designers are stepping into the fray and succeeding in their place. This is certain to be seen as a threat to programmers, but it need not be. Programming takes great skill, talent and creativity. It is not diminished by design; it is exalted by it. Programming won't diminish in importance, any more than surgery is diminished by epidemiology. If the technical management in our industry understands this, they can make a place in the development process for design, and much of the rancor in the struggle can be avoided. In sequence, design comes first, then user testing, then programming, then more design and more user testing, then more programming. Design leads the parade. Programming brings up the rear. Programmers sense that they have a lot to lose, they don't like it, and they will fight it. I don't blame them, but they must relinquish a seat at the table in order to continue eating well.

User interface designers will not come from the ranks of programmers but will be very technically savvy people. Non-technical people cannot imagine the

wonderful new things that computers can do for us. Non-technical people will not understand the delicate balance between a CPU with time on its hands and one rushing to complete ten million instructions before the user's next keystroke. The non-technical people will have our computers treating us in the same lousy way they already do but with prettier pictures. It is not obvious what computers can do for us. It takes natural talent, skill, training and experience.

Well, now that I've said that, you are probably asking "trained where?" and I must admit I don't have a ready answer. Academia has a hard time with processes that cannot be tested scientifically, so design doesn't compete well in ivy-covered halls against usability testing. This is why I believe that the majority of tomorrow's software design leaders will come from industry rather than from college.

Most universities that I am familiar with are teaching empirical user-centered usability testing more than they are creativity-based design. But things are bound to change, and more and more designers are appearing every day. Many of them come from the ranks of technical support people, quality assurance testers, technical writers and other professions that are often viewed as subordinate to programming. User interface design is not an art, like painting and sculpture, but its beating heart is creativity. Ironically, so is programming's.

"I'm mad as hell, and I'm not gonna take it anymore"

In the early '70s, Detroit said that they gave the American consumer just what they wanted: big, heavy, powerful, gas-guzzling, chrome-encrusted, fin-studded symbols of post-war largesse. Then Japanese and German cars built with very different sensibilities became available. The American public bought them by the millions, and Detroit was devastated by the foreign invasion. To their credit, American auto manufacturers cleaned up their act. But one thing you never hear from Detroit anymore is that old saw about "what the consumer wants." We've learned not to trust a statement like that.

The conundrum is that the American automobile consumer of the early '70s was perfectly happy with those gas-guzzling behemoths from Detroit. They didn't realize that they could do better until they had their noses rubbed in it by the availability of something better, cheaper and more environmentally kind.

The American software consumer today is just like the typical auto consumer of the '70s. The quality and usability of all of the software available is about the same: Big, monolithic, implementation model, only available from one or two sources who make design decisions based on what is good for the CPU instead of what is good for the consumer.

I want to show today's software consumer that things can be different. I want him to see the potential for software that is designed to help him reach his goals instead of programmed for the convenience of his computer. I don't want the domestic software industry to undergo the same painful upheaval that Detroit did in the '70s. I don't want to see the wind banging through the Microsoft campus like it does in abandoned Rust Belt steel mills. Before that happens, I want consumers to get angry and demand better. I would like to see the user community rise up in protest at the sorry state of software. I'd like to hear them protesting in the streets and picketing Oracle, Novell, Lotus, Apple and Microsoft, chanting—as viewers did in Paddy Chayevsky's classic screenplay *Network*—"I'm mad as hell, and I'm not gonna take it anymore."

Reference Section

Axioms

A dialog box is another room. Have a good reason to go there.

A gallon of oil won't make a bicycle pedal itself

A multitude of gizmo-laden dialog boxes doth not a good user interface make

A rich visual interaction is the key to successful direct manipulation

A visual interface is based on visual patterns

Accepting bounded data into unbounded gizmos is an important source of user dissatisfaction

All idioms must be learned. Good idioms only need to be learned once.

Allow input wherever you output

Any command is a working set candidate

Ask forgiveness, not permission

Audit, don't edit

Consistency is not necessarily a virtue

Directly offer enough information for the user to avoid mistakes

Disks and files make users crazy

Disks are a hack, not a design feature

Do, don't ask

Don't hamper primary markets by serving secondary markets

Don't make the user look stupid

Don't put might on will

Don't stop the proceedings with idiocy

Good user interfaces are invisible

Hide the ejector seat levers

Hot, simple and deep

If it's worth asking the user, it's worth the program remembering
Imagine users as very intelligent but very busy
It's not your fault, but it's your responsibility
Make errors impossible
Make everything reversible
Never bend your interface to fit a metaphor
Never make the user ask to ask
No crisis inside a computer is worth humiliating a human
No matter how cool your interface is, less of it would be better
Nobody wants to remain a beginner
Obey standards unless you've got a darn good reason
One user's excise task is another user's revenue task
Optimize for intermediates
Provide an escape from dragging, and inform the user
Purchase the right software then buy the computer that runs it
Put primary interaction on the primary window
Questions aren't choices
Show don't tell
Sovereign users are experienced users
The computer does the work and the user does the thinking
The customer is always right
The goal of all software users is to be more effective
Things that behave different should look different
Transliterated mechanical models are always worse on computers
User interface design is not guesswork
User interface is not just skin deep
User interfaces that conform to implementation models are bad
User testing can never substitute for user interface design
Users don't make mistakes
Users get humiliated when software tells them they failed
Users make commensurate effort
Users would rather be successful than knowledgeable
Visually hint at pliancy
Visually show what, textually show which

Design Tips

All dialog boxes should have caption bars
Any program that demands precise alignment must offer a vernier
Any scrollable drag-and-drop target must autoscroll
Build function controls into the window where they are used
Build the program to run on only one platform
Button-down means propose action, button-up means commit to action over gizmos
Button-down means select over data
Cancel drags on chord-click
Debounce all drags
Dialogs break flow
Dialogs should be as small as possible, but no smaller
Disable menu items when they are moot
Don't put close boxes on modal dialogs
Don't stack tabs
Don't use bang menu items
Don't use dialogs to report normalcy
Double-click means single-click plus action
Eliminating excise makes the user more effective
Error message boxes stop the proceedings with idiocy
Every text item in a list should have an identifying graphic icon next to it
Give modeless dialog boxes consistent terminating commands
Have a reason for each idiom
Indicating pliancy is the most important role of cursor hinting
Make selection visually bold and unambiguous

Menus and dialogs are the pedagogic vector
Never change terminating button captions
Never create a system modal dialog box
Never scroll text horizontally
Never use sustaining dialogs as error messages or confirmations
Never use terminating words in dialogs
Offer bounded gizmos for bounded input
Offer OK and CANCEL buttons on all modal dialog boxes
Offer shortcuts from the help menu
Offer the user a gallery of good solutions
Parallel visual symbols on parallel command vectors
Prepare for the probable case
Put terminating buttons on untabbed area
Show validated entry gizmos with a different border
Single-click selects data or changes the gizmo state
The drag cursor must visually indicate the master object
The drop candidate must visually indicate its dropability
The program must inform the user when it gets stupid
The program should be designed expressly for the target platform.
The program should perform optimally on hardware that doesn't exist yet
Toolbars provide experienced users with fast access to frequently used functions
Use COLOR_HIGHLIGHT and COLOR_HIGHLIGHTTEXT to show selection
Use cursor hinting to show meta-key meanings
Use object names in property dialog caption bars
Use verbs in function dialog caption bars
Users don't understand boolean
Visually differentiate modeless dialogs from modal dialogs

Index

Dear Reader,

You have heard from me, now I would like to hear from you. Please write or e-mail me and tell me what you thought about this book, and about software design in general. Let me know your contact information by returning this form, and I will keep you informed about our products, services and activities.

My consulting company, Cooper Software Inc, designs state-of-the-art user interfaces for companies large and small. We help our clients to improve their existing products or to create new ones. Cooper Software also offers seminars and training in user interface and conceptual software design.

I look forward to hearing from you.

Sincerely,

Alan Cooper, *President*

- ❑ Put me on your mailing list.
  - ❑ I want to know more about your seminars and training.
  - ❑ Tell me about your consulting services.
  - ❑ Keep me posted about future publications.

Name _____ Title _____

Company _____

Address _____

City _____ State _____ Zip _____

Phone _____ Fax _____

e-mail _____

How can Cooper Software help you?

COOPER SOFTWARE INC

Cooper Software Inc
POB 4026
Menlo Park CA 94025

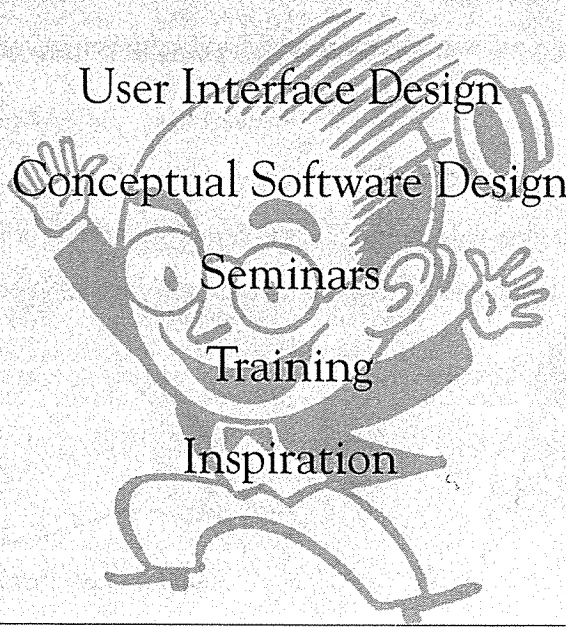
design@cooper.com
1-800-928-3374
Fax 1-415-322-8001

Visit Our Webpage At:
http://www.cooper.com

WHAT

COOPER SOFTWARE

CAN DO FOR YOU



User Interface Design

Conceptual Software Design

Seminars

Training

Inspiration

COOPER SOFTWARE INC

Cooper Software Inc
POB 4026
Menlo Park CA 94025

PLEASE PLACE POSTAGE HERE

Cooper Software Inc
POB 4026
Menlo Park CA 94025

IDG BOOKS WORLDWIDE REGISTRATION CARD

ISBN Number: 1-56884-322-4

Title of this book: About Face: The Essentials of User Interface Deisgn

My overall rating of this book: ❑ Very good [1] ❑ Good [2] ❑ Satisfactory [3] ❑ Fair [4] ❑ Poor [5]

How I first heard about this book:

❑ Found in bookstore; name: [6]
❑ Book review: [7]
❑ Advertisement: [8]
❑ Catalog: [9]
❑ Word of mouth; heard about book from friend, co-worker, etc.: [10]
❑ Other: [11]

What I liked most about this book:

What I would change, add, delete, etc., in future editions of this book:

Other comments:

Number of computer books I purchase in a year: ❑ 1 [12] ❑ 2-5 [13] ❑ 6-10 [14] ❑ More than 10 [15]

I would characterize my computer skills as: ❑ Beginner [16] ❑ Intermediate [17] ❑ Advanced [18] ❑ Professional [19]

I use ❑ DOS [20] ❑ Windows [21] ❑ OS/2 [22] ❑ Unix [23] ❑ Macintosh [24] ❑ Other: [25] (please specify)

I would be interested in new books on the following subjects:
(please check all that apply, and use the spaces provided to identify specific software)

❑ Word processing: [26]
❑ Spreadsheets: [27]
❑ Data bases: [28]
❑ Desktop publishing: [29]
❑ File Utilities: [30]
❑ Money management: [31]
❑ Networking: [32]
❑ Programming languages: [33]
❑ Other: [34]

I use a PC at (please check all that apply): ❑ home [35] ❑ work [36] ❑ school [37] ❑ other: [38]

The disks I prefer to use are ❑ 5.25 [39] ❑ 3.5 [40] ❑ other: [41]

I have a CD ROM: ❑ yes [42] ❑ no [43]

I plan to buy or upgrade computer hardware this year: ❑ yes [44] ❑ no [45]

I plan to buy or upgrade computer software this year: ❑ yes [46] ❑ no [47]

Name: Business title: [48] Type of Business: [49]

Address (❑ home [50] ❑ work [51]/Company name:)

Street/Suite#

City [52]/State [53]/Zip code [54]: Country [55]

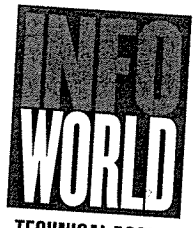
❑ **I liked this book!** You may quote me by name in future IDG Books Worldwide promotional materials.

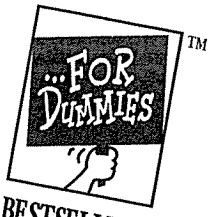
My daytime phone number is _____

IDG BOOKS WORLDWIDE™
THE WORLD OF COMPUTER KNOWLEDGE™

❑ YES!

Please keep me informed about IDG Books Worldwide's World of Computer Knowledge. Send me your latest catalog.





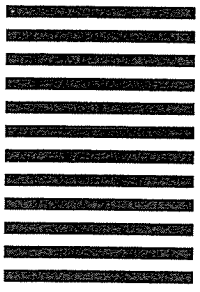
BESTSELLING BOOK SERIES FROM IDG

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 2605 FOSTER CITY, CALIFORNIA

IDG Books Worldwide
919 E Hillsdale Blvd, Ste 400
Foster City, CA 94404-9691



ABOUT FACE

THE ESSENTIALS OF USER INTERFACE DESIGN

"Alan Cooper is the 'Miss Manners' of software design... My advice is to buy two copies—autograph the second, and send it to an engineer at Microsoft."
— Paul Saffo, Director, Institute for the Future

"*About Face* defines a new interface design vocabulary that speaks to programmers in their own terms. We have come a long way from the time when there were just modal (bad) and modeless (good) interfaces, and this book reflects that progress."
— Charles Simonyi, Chief Architect, Microsoft Corp.

COOPER

Dear Reader,

This book has a simple premise: If achieving the user's goals is the basis of our user interface design, then the user will be satisfied and happy. If the user is happy, he will gladly pay us money, and then we will be successful.

To those who are intrigued by the technology — which includes most of us programmer types — we share a strong tendency to think in terms of functions and features. This is only natural, since this is how we build software: function by function.

The problem is that this isn't how users want to use software. Developers are frequently frustrated by this, because it requires us to think in an unfamiliar way. But after the initial strangeness wears off, goal-directed design is a boon — it is a powerful tool for answering the most important questions that crop up during the design phase:

- What should be the form of the program?
- How will the user interact with the program?
- How can the program's functions be most effectively organized?
- How will the program introduce itself to first-time users?
- How can the program put an understandable and controllable face on technology?
- How can the program deal with problems?
- How will the program help infrequent users become more expert?
- How can the program provide sufficient depth for expert users?

In *About Face*, you'll explore new ways to look at what you work with every day, learning how to create workable designs in the real world, on a real deadline, inside a real budget.

Sincerely,

Alan Cooper
President
Cooper Software Inc.

About the Author

Alan Cooper is one of the most respected software designers of our time. He is the winner of the Microsoft Windows Pioneer Award for his work in designing Visual Basic. He is also one of the most outspoken critics of how the software industry goes about building the interface between products and people. His ten-year-old software design consulting company, Cooper Software Inc., is based in Menlo Park, CA.

Technical Review by Neil Rubenking, Technical Editor, *PC Magazine*

FIFTH ANNIVERSARY IDG BOOKS WORLDWIDE
Leading the Knowledge Revolution™

IDG Books Worldwide, Inc.
An International Data Group Company
Foster City, CA 94404

LEVEL
Beginner to Advanced

COMPUTER BOOK SHELVING CATEGORY
Interface Design/Windows/ Programming

PROGRAMMERS™ PRESS

$29.99 USA
$39.99 Canada
£28.99 UK