This essay deals with a very down-to-earth topic: the things computer programs or systems say to people—in particular, computer messages for users. It is the product of the author's experience as a programmer and technical writer and editor. It puts together a lot of common-sense insights into the philosophy of creating good computer messages, how people think and feel around computers, how to analyze the situations in which people need a message, what to say in a message and how to say it, why imagination is invaluable for creating and evaluating messages, what technical questions must be answered in order to design and build a program or system that can talk effectively to people.

How a computer should talk to people

by M. Dean

Something that makes a computer easy or hard for us to use is the way it "talks" to us. Good messages help make it easy for us to use; bad messages help make it hard.

One reason that some computer programs or systems contain bad messages may be that "message" has come to mean a terse one-liner that people *are not expected to understand* without an explanation. In fact, there are guidelines for preparing documentation to *explain* messages. And manuals are written to reveal what messages often do not reveal—their meaning.

People want a computer to provide messages that explain themselves, that say what they mean. In fact, psychologically, *the meaning is the message*. A message whose meaning has to be explained does not communicate—it fails as a message.

How do we ensure that a computer's messages are useful to the people who receive them?

From my five years as a programmer and systems engineer working with customers, and from the last eleven years as a technical writer and editor of software publications, I have learned some things about

how a computer ought to talk to people. My main job as an editor is to playact: to imagine myself as the computer user, to imagine how users will respond to a program and its documentation. I try to expect what users will expect and to notice the patterns they are likely to form (or not be able to form). I try to spot passages people will probably have to re-read, and I register any empathetic feelings of uncertainty.

My experiences have suggested some ideas about how to produce good messages. In this essay, I formulate some conclusions and share them with you. I hope you will find them useful.

I have noticed how programmers usually try to create good messages. After their programs are practically finished, they get someone to edit and reword the messages that have sprung up like weeds during program or system development. Thus, they try to ensure that essentially unplanned messages are concise, grammatical, consistent, and understandable. But these qualities do not guarantee that the messages are also relevant, specific, timely, and helpful. I have concluded that you cannot edit and reword the second set of qualities into messages that already exist.

To create messages with both sets of qualities, there are a number of things that have to be done. In the rest of this essay, let us consider them, roughly in the order they should be done:

- Set human goals for messages.
- Apply psychology in writing messages.
- Write messages that accommodate intended users and their situations.
- Playact to evaluate the messages for usability before coding.
- Edit the messages for appropriate language.
- Design the computer program or system to produce the messages.
- Test the messages along with the running code.

Set human goals for messages

People want the computer to accommodate them—not vice versa. Accordingly, what messages should a particular program send? I recommend that you answer that question before you build the program; decide what messages to send, what information to put in them, and how to present that information. To design a program whose messages are relevant, specific, timely, and helpful, I believe we must commit ourselves to

- Being tolerant of "user errors."
- Helping people correct errors as easily as they make them.
- Giving people control over the messages they receive.
- Not making messages arbitrarily short.
- Identifying the messages that people need.

Be tolerant of "user errors"

In their book on humanized input, Gilb and Weinberg[1] argue that programs should correct user errors for which corrections are safe and highly probable. I agree with the authors that programs should be much more tolerant than they usually are of errors we make when we try to communicate with a computer. Gilb and Weinberg identify several levels of program reaction to human input:

levels of computer reaction to human input

Immediately usable input. The authors note that, of course, the program goes ahead and processes such input.

Input that is usable after an obvious, or highly probable, correction. The authors recommend that the program correct the input, optionally inform the person, then go ahead and process the corrected input. (The program should probably ask for a go-ahead when processing will have results that are difficult or impossible to undo.)

Input that is usable after an unreliable correction. The program should correct the input, ask the person for a go-ahead or alteration, then go ahead and process the corrected input (or not if the person says not). (The person's response might be to choose among possible alternatives the program has identified, or reject them all, or propose another alternative.)

Unusable input. When the input is unintelligible or clearly wrong and uncorrectable, the program should tell the person why it cannot proceed, and suggest something helpful.

Where the designer draws the line along this scale becomes a design point for the program. I have noticed that often the line seems to be drawn between "immediately usable" and "usable after obvious correction." That is, input not immediately usable is rejected for the person to correct. For example, assume that the input to a program is supposed to be a measurement rounded off to the nearest whole number. Let us say we measured 10.3 and therefore entered "10.0". The program, instead of accepting "10.0", tells us

Incorrect syntax. Reenter the number.

The program does not even say how to make the syntax correct. It probably has not investigated precisely what is "wrong" with "10.0". If it had, it might have ignored the ".0".

I have heard two objections to undertaking to build error-correcting programs: "It is often impossible to know what the user intended," and "Correcting user errors would double the amount of code." These objections are serious; the problems are difficult. But they are being tackled. For example, three Carnegie-Mellon researchers have described their attempts to build a "tool-independent system which

can serve as the user interface for a variety of functional subsystems."[2] After standard error-correcting program modules are written, programmers will not have to do error-correction anew for each program.

Help people correct errors as easily as they make them

In the auditorium in one of the buildings at IBM's Poughkeepsie location there is a lectern which can have the incline of its top surface, or desk, electrically raised or lowered. There is only one button for a speaker to push, and pushing it may either raise or lower the incline of the desk. But the desk, once started, must go all the way before reversing direction.

Once when I spoke at this lectern, I wanted the incline of the desk raised, so I pushed the button. Of course, the desk started lowering. I held the button down for ten seconds or so while the incline of the desk was first lowered, then raised to about where I wanted it. But I had "erred" by holding the button down a split second too long, so that the desk was now a little higher than I wanted it.

I would have liked a second button to press for a split second of corrective lowering. But, instead, I had to hold the one button down for another ten seconds while it rose fully, then lowered to . . . about where I wanted it!

ease of correction

Ideally, we should be able to correct an error with no more effort than it took us to make the error in the first place. When a program cannot let us correct an error literally as easily as making the error, it should at least help us out. For example, we can usually modify what we said more easily than resaying the whole thing, hence the popularity of terminal keys that redisplay previous input. Why not design a program to give back the input for modification and indicate somehow where the modification seems to be needed? In any case, we can show the input in error along with the message that refers to it.

Or perhaps a program can think of three things we may have meant. Does the program engage us in a dialogue to find out which we meant, if any? Possibly. Possibly not. It may be easier for us to resay something short than to "discuss" the problem with the computer.

Give people control over the messages they receive

From interactive programs, different people want different amounts of information, and the same people want different amounts at different times.

Certain messages for information—defaults assumed and corrections made by the program—are most useful when we are learning. These messages lose their usefulness with repetition. We become so familiar with a program after a while that we do not read the

messages anymore. Why doesn't the program let us turn these messages off and turn them back on when we want? And, because we know the use of some parts of a program better than other parts, the program might let us turn messages off selectively by program function.

One reason for not letting us turn messages off is that occasionally an unpredictable message shows up. But if we are ignoring messages, we will not see it anyway unless the message strikes us as being different. Maybe a program whose messages are turned off could alert us when there is an unusual message, or deliver the message anyway. ("Unusual" is relative to the "usual" messages we have become insensitive to; the program would have to remember what messages it has been sending.)

Because message length varies and can be several lines long, it may be impractical or even unacceptable to us for an interactive program to always provide the entire message immediately. Our acceptance can vary with the speed of our terminal and its method of display and with our expertise. A program might let us choose a detail level for messages generally (for example, it could give us the option of a short form or a long form), then let us get more information when we need it.

Techniques for doing this have become known as *help facilities*, though their helpfulness is often limited because help panels usually give general reference information rather than specific advice for the given situation, and they obliterate the screen that contains the input that is in error.

A solution would seem to call for "helps" to provide small increments of additional information that (a) are specific and relevant to the input that is in error and (b) can be displayed along with the input.

Do not make messages arbitrarily short

Though "brief" and "concise" mean "no longer than necessary," I often hear programmers talk as though they mean "no longer than some arbitrary length." But some useful messages simply will not fit into 58 spaces—or whatever arbitrarily short limit—because to fit them in, you have to omit something essential.

One solution is to provide a variable space for messages. When necessary, it would expand temporarily to accommodate two or more lines. A product at the lab where I work that has used this solution lets messages take up to half the screen, but a programmer reported to me that "actually, none is more than five or six lines long."

For messages to survive an arbitrarily short limit (such as one line), you need some way to provide additional information when it is

necessary. How do you do that? What information do you provide in the one line—a shorthand version of "everything"? a summary? a salient detail? And what additional information can the person request—a "longhand" repetition of the shorthand? the items that were summarized? the rest of the details? Do you let the person know when there is additional information, or can the person always request it, whether there is any or not?

These are thorny questions of display design. For example, the arbitrarily short message-display area may very well be too short for the additional information. Surely we must not show a multiline message a line at a time. People need to see it whole and, where appropriate, along with the input or other information to which it refers.

If we "solve" these thorny display problems by ignoring them and stuffing all messages into an arbitrarily short space, we will create many meaningless messages.

Identify the messages that people need

A program designed well for people provides messages to serve their interests. Before you design the logic and start coding, you can identify those interests; that is, you can define the "user interface."

The usual procedure in creating programs is to design and document the way the program is *supposed* to work and the way people are *supposed* to behave. What the program was not designed to handle becomes by default an exception, to be discovered during coding. I have noticed that, unfortunately, ways devised *ad hoc* to handle these exceptions tend to be for the convenience of the program and not that of the people.

plan for unpredictable errors

But when you design programs for human use, not only should you define what happens when there are no errors, either in the program or in what people do, but you should also anticipate the kinds of mistakes people are apt to make, and plan how you will deal with unpredictable program errors (and user errors that you could not anticipate):

- You can easily anticipate the messages that reflect error-free operation. You should write those messages and document them in your specifications.
- Anticipating the mistakes people are likely to make is more difficult. But by anticipating them, you can act to avoid them or to minimize the disruption they entail. Classify these mistakes according to a scheme like that of Gilb and Weinberg[3] and see what messages are called for. Errors the program can correct call for messages noting the correction, or asking for clarification or go-ahead. Uncorrectable errors call for messages that say what is

wrong, why, and what to do. You should write these messages and document them in your specifications. (You probably cannot anticipate *all* user errors. Some you will discover only during coding. But if you have a strategy for responding to user errors, the messages for unanticipated errors too can be relevant, specific, timely, and helpful.)

- Of course, you cannot predict all the system errors and adverse conditions that can occur; they will result from the unique complexity of your program's logic. But you can establish in advance a strategy for what the program will tell (a) users, (b) people responsible for keeping the system available, and (c) program diagnosticians responsible for investigating and fixing errors.

Summary of setting human goals

We have discussed the need to set human goals for messages in order to commit ourselves to being tolerant of "user errors," helping people correct errors as easily as they make them, giving people control over the messages they receive, not making messages arbitrarily short, and identifying the messages that people need. Now, let us look at some psychology for writing messages.

Apply psychology in writing messages

A message that is delivered to the right person on time can still fail for many reasons: the words are unfamiliar, something is missing, there is too much detail, the program's point of view is not the same as the person's, the facts are vague. It is essential to learn as much as you can about the people who will use your program. They may not share your interests in computers and programming. Their vocabulary may be smaller than yours or, if it is larger, they still may not know the meaning of specialized words such as "byte."

basic lessons

I have read a number of psychology textbooks and practical books about how to read, how to study, how to remember, how to think visually, how to solve problems. This reading has strengthened some lessons of my own experience about how people think and feel:

- Our expectations govern the way we react in various situations.
- We need to see how all the pieces fit together.
- We do not want to have to re-read something to figure out what it means.
- We learn about and use a computer more effectively when we feel at ease.

As I see it, the job of the program designer and message writer is to apply these principles to their designing and writing. The following observations about how to do that are adapted from a recent article of mine.[4]

Anticipate people's expectations

We are disposed by our expectations to act a certain way in a given situation. When a message does not match our expectations, interference results.

Expectations result from our past experiences and our future needs. Our past experiences include our training (or our lack of training) in the use of computers and computer vocabulary. They include our previous use of a program, reading the manuals about it, receiving previous messages. They include what we have just now been doing or attending to.

Our past experiences also include our cultural backgrounds. One implication of this is that American messages should not simply be translated literally into Italian, or Japanese, or Pakistani. Cultural differences may call for message redesign by someone who not only knows how to use a foreign-language dictionary, but also understands the two cultures in question.

Our future needs are what we would like to do next. They are essentially our desire to accomplish the task for which we are using the computer. A message that overlooks our practical situation forces us to determine how it applies to what we are trying to do.

matching expectations

People who can easily write good messages do not necessarily know how they do it. In an attempt to understand why these people are successful, I have discovered that they are sensitive to three aspects of a message when they are trying to make it match people's expectations: (1) the *meaning* to be conveyed, (2) the *information* selected to convey that meaning, and (3) the *language* chosen to present that information.

For example, a sign at a botanical garden might need to convey to people wandering off the path the meaning: "Do not pick the flowers!" To convey this meaning, the sign might include the information that "the plants are rare and hard to replace," or that "pickers are subject to a $500 fine." As to language, the sign might actually say:

The flowers are not for picking. Picking can result in a $500 fine.

or

Do not pick the flowers, or you will be fined $500.

For a message to match people's expectations:

- The *meaning* to be conveyed must be relevant to their situation. If it is not, either do not formulate the message at all, or else identify the people to whom it is relevant. (The botanical garden really only wants to dissuade potential pickers; it does not want to distract nonpickers from enjoying the garden.)

- The *information* selected to convey the meaning must also be relevant to the people's background and needs. If it is not, add what is required, and get rid of what is extra. (The information about the rarity of the flowers might suffice for people simply unaware of their value, but only the threat of a fine might dissuade those who do not care whether or not the flowers are valuable.)
- The *language* chosen to present the information must also be appropriate. The words, and the way they are put together, must be familiar. If they are not, change the vocabulary, rearrange, rewrite. (A practiced writer can help here. If you have a relevant meaning and have selected the needed information, a wordsmith can help you find the appropriate language.)

A message that results from a person's action must relate to that action. But what about something like this: We have entered a command whose operands are supposed to be separated by commas and ended with a period, like items listed in a sentence. But we have ended the list with a comma:

copy fromfile, tofile, truncate,

What is wrong with the following message?

Missing operand in the COPY command.

From the program's point of view, this is one way to understand what is wrong with the input. But *we* do not understand it that way; the message does not conform to what we expect.

A message can make us act in a way that is different from usual by changing our expectations. Unfortunately, the change can be for the worse, as well as for the better. Because the message in the preceding example mismatches our expectation (to end a list), it distracts us; we may begin now to look for something missing. The following revision allows for the fact that, despite the final comma, the input makes sense and is reasonable:

"copy fromfile, tofile, truncate," is interpreted as
"copy fromfile, tofile, truncate."
Proceed or modify? Type "p" or "m," then press ENTER.

Below is another example of misdirection by a message. A "little white lie" is the culprit; can you detect it?

DISPLAY not processed, because file DEPTROLE not found.
Use the correct name, or get authorization to see DEPTROLE.

The phrase "not found" is a distortion. For, as the message implies, the file we want to display may have been found, but we do not have the authority to look at it. This message would confuse some of us.

I have seen enough examples of such distortions in messages to think they must be fairly common. We ought to make them less common.

Help people fit the pieces together

People try to fit things together to make sense of them. We want to form a single idea or pattern from all the pieces, the best pattern we can form under the circumstances. To do this, we group pieces that have affinities. For example, we often assume at first glance that because one thing looks like another, the two things are really alike. Or, we may overlook something extraneous:

The meaning of a message is the point the
the message makes

or invent something:

p gr m g

Raw facts are less important than the meaning they add up to. That this is true we can see from the observation that other facts would do as well, so long as they convey the same meaning. Do not just give people raw facts and leave it to them to guess what the facts mean, especially when people might guess wrong. (By the way, did you make "pregrooming" out of "p gr m g"?) For example, what might this message mean?

You have been logged on 28 minutes.

It might mean that you are going to be forced off in two minutes, unless you log off voluntarily. Of course, people who are aware that they cannot stay logged on for more than 30 minutes may understand that. But not everyone is aware of this curious practice.

proven methods to fit pieces

Here are some proven ways to help people fit the pieces together:

Proceed from the general to the specific. For example, look at the sentence that leads into these paragraphs ("Here are some proven ways. . ."). It announces that the specifics to follow are instances of "proven ways." A message with an unannounced—or poorly announced—list would confuse some people:

ABC0001 [list of parameters]

People may not see the significance of the list, or how the items are related. Here is the list again, with a headline to shed some light on the significance of the list:

ABC0001 Data could not be read.
[list of parameters]

Proceed in parallel, treating like ideas alike. That is, be consistent within a message and among messages. Information presented one way sets up an expectation that similar information will be presented the same way in the future. For example, if I see several error messages that first explain my error, then say what to do, I come to expect the order: explanation followed by suggested action. A message with the reverse order will thwart my expectations.

Indicate when an action is finished. People like to know that they have been heard and what the result of their action was. They do not want to be left hanging. Their input is not an isolated action. It is an attempt to communicate, and it is not finished until they know they have been successful. Do not leave them in limbo. Let them know what is happening as a result of their action.

Do not force people to re-read

When we read, we have to make sense of words, phrases, clauses, sentences, and paragraphs as we go along. We would like to get the meaning of a message the first time through. But sometimes we cannot, because we get overloaded by too many words or phrases whose interconnections we cannot understand. Consequently we are not able to fit them all together into a single idea; we have to re-read:

The operand that has three positional parameters ("a", "b", and "c"), any one, but only one, of which can be omitted by coding a lone comma for it ("a,b,,", "a,,c", or ",b,c"), can itself be omitted by. . .

There seems to be a very small limit (about five[5]) to the number of words, phrases, and clauses whose meaning we can leave unresolved at one time. If that number is exceeded before we figure out the meaning of a sentence or a paragraph, we may have to re-read it to put the pieces together.

When we re-read because too many unresolved items are dangling, we try to combine some items; this reduces the number of items we have to deal with. For example, a dozen binary digits ("010011110101") are too many to remember in one pass. In subsequent passes, we can group the bits ("0100 1111 0101") and represent each group with a single placeholder ("4F5"). The information content is the same, but more efficiently represented.

proven ways to avoid re-reading

But why leave this sort of thing to our readers? As writers, we can do it for them when it seems to pose a problem. Here are some proven ways to help readers avoid having to re-read:

Write in terms of the familiar. Refer as much as possible to things people know about, in terms they are used to seeing. They can interpret and combine familiar things more readily than unfamiliar ones. For example, if I have requested "convert osfile to dosfile," the progress message, "Constructing header labels," may not mean much if I am not familiar with the internal substeps of conversion. And if much time has gone by, I may not remember I was converting. Better: "Conversion in progress: step 2 of 5 complete (header labels have been constructed)."

Provide concrete specifics. We are most comfortable with concrete specifics; we immediately respond to things we can actually see,

touch, or hear, or can easily imagine. We may not be able to do this for an abstract term ("performance degradation"). Or if we can, we may imagine something different from what was meant. (We may imagine that "performance degradation" means "errors in processing," whereas the writer intended "twice as long to respond.") These uncertainties prevent us from speedily resolving meanings and fitting all the pieces together. So, write concretely and, when exactitude is important, be precise.

Show the relationship among things. When we cannot see the relationship among things, we cannot combine them to reduce the number of items we have to deal with. When the relationship is not evident, a single idea cannot emerge. For example, what is wrong with this message?

No space available to sort; press the PF1 key.

You probably understand "no space available to sort" and you may know which is the PF1 key, but you probably do not see the connection. Will pressing the key get space, or cancel your sort request, or what?

When our readers feel secure—because we use familiar and concrete terms and show the relationship among them—they can speedily resolve meanings and fit all the pieces together. They probably will not have to re-read.

Put people at ease

We learn about and use a computer more effectively when we feel secure and experience success with it. What helps us feel at ease with a computer program or system?

For me and for most people that I have talked with, the most useful thing is to have a *reliable, stable concept of the program or system*, with little uncertainty about what it will do (it is predictable), or at least a high expectation that it will not do us in (it is benevolent).

People are going to form some concept or other of our program. We should help them form a helpful one. How?

The program should provide adequate information about itself, such as defaults it is assuming, corrections it is making in input, and any other assumptions it is making about the input (that is, about people's intent). Showing defaults that are assumed, for example, lets people know what options govern an operation. This reminds them indirectly that they will have to specify different options whenever they want the operation to go differently. Showing corrections the program has made can teach people how to prepare input correctly.

Friendliness also helps people feel at ease. And it is not difficult to be "friendly."[6] It does not require special gestures or mannerisms. Just

provide a helpful message—one that lets people know what is happening now so they can predict what will happen next, or one that lets people actually control what will happen next by their response to the message.

The main way we are unfriendly to other people is to ignore them. Another way we are unfriendly is to give an obscure message, because it can threaten people who are already insecure. They may think they are incapable of understanding (when, in fact, the message cannot be understood).

Various negative tones or actions are unfriendly: being manipulative, not giving a second chance, talking down, using fashionable slang, blaming. We must not seem to blame the person. We should avoid suggesting that the person is inadequate. Phrases like "you forgot" may seem harmless, but what if a computer said this to you four or five times in two minutes? Anyway, the person may disagree, so why risk offense?

Nothing succeeds in overcoming uncertainty like *success*. Early successes make for effective and efficient learning. They build our confidence.

Ideally a program should make it hard for us to fail in using it, which is another reason for Gilb and Weinberg's recommendation to make programs more tolerant of human mistakes.[7] If a program itself corrects people's mistakes, persons learning the program will experience fewer failures; they may feel more competent and learn more quickly.

Summary of psychology for writing messages

We should make use of what we know about how people think and feel when they use computers: Their expectations determine how they react in various situations. They need to see how all the pieces fit together. They do not want to re-read anything. They learn about and use a computer better when they feel at ease. With this in mind, we are ready to consider the actual writing of messages in particular situations.

Write messages for the audience and the situation

How are we to discuss specific types of messages? What are the "types"?

I have observed that programmers use terms such as the following when they identify types of messages: confirmations, prompts, information messages, warnings, interrogations, operator messages, and error messages. These terms primarily reflect different purposes. The

sole exception is the "operator message," which is an example of a message categorized according to who gets it.

Categorizing messages by audience has its place; it indicates where we should send them. For example, a program might generate more than one message: one for the primary user and a related message for a system administrator, console operator, system auditor, or program diagnostician. Audience categorization also reminds us to notice individual differences among people. For example, the reading ability of one audience, or its familiarity with vocabulary, may be significantly different from that of another audience. If so, we should certainly write messages accordingly.

But I think that distinguishing by purpose is more helpful for writing messages, because what we say in a message depends first on its purpose. What is a particular message supposed to accomplish? Why should a person receive it in the first place? This way of putting it directs our attention to the need of the person who will receive the message. First we identify the need, then figure out how to satisfy it.

interactive messages

I believe that people who interact with a program need messages of the following types:

Report on the program's reaction to input. Tells us (a) that processing is finished and what the results are, (b) that progress is being made, or (c) that input is rejected (for some reason beyond our control).

Report on the program's assumptions about input. Tells us, for example, defaults the program has assumed, corrections the program has made in our input, or the program's interpretations of our intention.

Report on a program error or adverse condition. Alerts us that our processing is, or may be, adversely affected.

Request for a go-ahead. Gives us a chance to say, "No, do not do that."

Request to choose among alternatives. Lets us choose, for example, among actions the program might take, options that will govern processing, possible corrections to input, or possible interpretations of ambiguous information.

Request for missing information. Lets us explicitly indicate information the program is to use.

Request for correction or clarification of input. Gives us responsibility for correction when the program cannot understand input well enough to interpret it confidently.

In planning to write a message of any type, first analyze the person's precise information needs in the situation—what should the message tell the person? What pieces of information must the person have?

Then, for each piece of information, identify a reliable source for the person. Does the person already know the information? Does the context supply it? Must the message supply it explicitly? Can the person infer it from the message?

Not everyone can get the information from context or by inference. To write a message, you must be aware of the differences among individuals who will receive it: people new to the program versus people quite familiar with it; people new to computers versus people who have used them a lot. It is wise to be generously explicit in the first draft of a message; it is much easier to subtract information from a message later than to add it.

Now, for each type of message, let us see how we analyze the information need and identify reliable sources.

Reports on the program's reaction to input

A program should always acknowledge human input, so that people are not left wondering whether the program heard them. Upon receiving valid input, a program might: (a) process it and produce results immediately, (b) have to delay processing it, or begin processing it but take a long time, or (c) be unable to deal with it then. Let us examine the messages for each of these three reactions.

Processing is finished. In this situation, people need to verify that the results are what they intended, so they can complete the transaction psychologically and go on to the next thing.

Sometimes the context provides this information well enough. For example, when I tell a program that I want to use a certain text editor to edit data, what immediately happens is this: The first nine lines of my data appear on the screen, with a highlighted line across the top that includes among other things the name of the data. Without receiving a message, I have "gotten the message"; I can see that what I requested has been done, and I know to proceed. But conceivably a novice might not see this and might wait until it seemed safe to proceed. In this case, "processing" is not so much finished as begun. A message like this would help the novice:

Proceed with editing.

To end the editing session I enter "file." I do not see what is happening, but in a few seconds I receive a message, "R," which in itself is unilluminating. But, from reading the manual, I have learned that "R" stands for "Ready." And, because I have ended an editing session many times (and my data has always been filed before), I take "Ready" in this situation to mean that the data has been filed.

Novices, however, would have no inkling of what "R" means. They would have no confidence that their data had been saved. Consequently they might edit the data again, to look and make sure.

Shouldn't a message say what has happened? "R" or "Ready" neither says that something is done, nor identifies what. In the case I have been describing, both novice and expert would be happier if they received a message such as

[dataname] filed. Editing ended.

There is a delay. When no result is available immediately, people need to know that their input has been accepted. Otherwise, they are uncertain whether or not to repeat what they said. Even veteran computer users become anxious when the computer stops reacting immediately to their requests. They start pressing buttons and "trying stuff."

As a delay continues, people need to know that something constructive is happening. Otherwise, they wonder whether or not to do something to get things rolling.

A progress message would help, issued after x amount of time has elapsed and the request still is not done. The optimum x for issuing a progress message varies; James Martin suggests times for "interim reports" in *Design of Man-Computer Dialogues.*[8]

A progress message should say that the program is doing the request, or that a substep of the total process has been completed. It should identify the request in terms familiar to the person who made it. It should not mention a substep the person is not expected to know about. Rather, for example,

Conversion in progress: step 1 of 5 complete.

This message lets the person know everything is okay and indicates more waiting is likely (four more steps). The message does not name step 1 (which might confuse someone who knows only that "conversion" is being done, and not what substeps it involves).

An alternative to issuing unsolicited progress messages is to let people interrupt to ask whether progress is being made. This alternative has two advantages over unsolicited messages: it is easier to program, and the computer is not slowed down by continually issuing progress messages. A "What's happening?" key could be provided for this.

The program cannot accept the input. When the program is having its troubles and is not accepting even valid input, people need to know this immediately in order to revise their expectations. People would like enough information to decide whether they should wait for the computer to straighten itself out, or go do something else.

Because the rejection of valid input is inherently frustrating, a generously informative message seems to be called for. Not an abrupt "Not accepted," or "Try anything but that." The program designer has an interest in reassuring users that the situation "really is not so bad, and does not happen often." Probably the best way to do this is to give people positive, practical advice about what to do. Tell them what *will* work (if anything), not just what will not work. Tell them when things will be back to normal, or how they can find out. For example:

Box busted. CE called. Will update status phone every half hour. No estimate of when system will be back up.

Reports on the program's assumptions about input

In using a certain program we may have a shaky concept of what is happening now and no idea what will happen next. We may be unaware of what the program is assuming about our input. A report from the program can alert us by revealing a conflict between what we think is going on and what the program thinks. A report can teach by correcting or filling in our idea. It can reassure us by supporting our idea. In this type of message the program might report on defaults it is assuming, a correction it has made in our input, or its interpretation of our intention.

A message about the program's assumptions must not only state the assumptions but also indicate that the program is acting on them. Otherwise, people will be uncertain about the significance of the message. For example, to show defaults assumed, a program might echo the input, with defaults added:

Input: format journal
Echo: format journal APPEARANCE (STYLE 1COL DUPLEX NO) BIND (10) DEVICE (3800N6) PROFILE (OURPROF) LIBRARY (OURLIB)

The meaning of the echo is not evident; it cannot be inferred logically from the message. Only people who had learned the convention would see the significance of the message: that the input ("format journal"), with the defaults shown, is being processed. The following message explicitly states this:

Processing FORMAT JOURNAL, with these defaults:
APPEARANCE (STYLE 1COL DUPLEX NO) BIND (10) DEVICE (3800N6)
PROFILE (OURPROF) LIBRARY (OURLIB)

Most of the defaults in this example are well represented, mnemonically, as defaults certainly should be. But even so, the message does not provide a tutorial.

If you think people might disagree with a program assumption, the program should state the assumption, then ask for a go-ahead. With numerous defaults, as in the example above, people may well want to change one of them. They should, of course, be able to do that without having to resay their input.

(In the actual implementation from which I adapted the example, the defaults are presented to me under the control of a text editor. To change a default, I use the editor, then store the defaults away.)

Reports on a program error or adverse condition

A bug or an adverse system condition poses information needs not only for users but also for operators and other support people, and for whoever diagnoses program failures.

different needs for different people

The needs of these different audiences are very different:

- Users, in order to adjust to adversity and form different expectations, need to know that their expectations are correct but, because of a program error, cannot be fulfilled. They need to know whether they can do anything to recover from the situation or correct it, such as inform a support person (who is not on line for immediate communication from the program). An example is the following:

  BACKUP processing cannot continue, because of a severe program error. Retry your BACKUP request after the operator has restarted the program.

  This message rightfully says nothing about program internals, about which the audience presumably knows nothing (and needs to know nothing).

- People who take care of the system for users need to know the extent of the problem and how urgently they need to act. They naturally know more about the system's resources than users do. They need information for the task of restoring those resources to users as soon as possible. Users are not aware (and normally need not be aware) of what that task involves.

  The people who support the users may or may not be on line. If they are not, a message might be sent to a log available to them. For example:

  The common work area is full. User work areas are filling up. Run the [name] utility, as described in the [product] *Administration Guide.*

- Diagnosticians need details about events inside the program when something goes wrong. To acquire those details they need indicators that would be identified in the program's diagnostic information. A message to them (or to their log) should include as many of those indicators as possible:

  Internal error in module HAAWXYZ, function INSQUIRT. . .

  This information would be useless and misleading to users and their helpers.

  This class of message is the hardest to anticipate. The programmers I work with say that these messages are numerous, although they are rarely issued. They even include messages for errors

where the program does not know what is wrong or what to do. Only a trained person can diagnose the problem.

Messages of this type are reports (rather than requests for user response), even though they do "call for a response" in the sense that the user may be advised to see a support person, and the support person may be directed to investigate or to take some corrective action. They are reports because user response cannot lead immediately to normal continuation of processing.

Requests for a go-ahead

A program should ask for a go-ahead when

- The action about to be done will probably have results that cannot be undone or will be hard to repair.
- The program's assumed default, error correction, or interpretation likely conflicts with the person's intention.
- The side effects of processing are likely not what the person intended.

In each of these cases, there is a chance the person will say, "No, do not go ahead." (If there is no chance the person will say no, a statement of the program's assumptions about input might be appropriate.)

inform people

In the situations listed, people need to know

- What has already been done, if anything?
- What action are people being asked to approve?
- What are the consequences of saying yes, and of saying no?
- How do people signify yes or no?

For example, if in processing input the program notices potential undesirable consequences, it might pause to ask something like

READ request temporarily suspended because another user has the record and may update it. Do you want to read the record anyway?

Some problems with this message:

- It does not identify the record, and the identity of the record might affect a person's decision.
- The message does not state either the consequences of reading the record anyway, or the consequences of saying no. The consequences of reading the record anyway are implied indirectly (if the other user updates it, this user will not see the "latest" version). But if this user says no, will the program wait until the other user is through with the record? If so, how long? (Could the user change a "no" to a "yes" later?) If the program does not wait, what *does* it do?

- The message does not tell how to give an answer. All of a program's go-ahead messages should follow the same practice on this point. They might include " 'y' or 'n' " or " 'y' to. . .; 'n' to. . ."

Requests to choose among alternatives

Listing alternatives shows people what is possible. The program analyzes a situation into its practical components for people and directs their attention to them. A program might give a choice among alternative actions the program can take, or alternative options that govern the requested processing, or alternative corrections of erroneous input, or alternative interpretations of ambiguous information. People should be able to reject all the stated alternatives, possibly in favor of another alternative, which they can specify.

In all of these situations, people need to know

- What is being done, or has already been done?
- Among what alternatives are people being given a choice?
- Why are they being given a choice?
- What are the consequences of each alternative?
- What are the consequences of rejecting them all?
- How do people signify their answer?

choosing among alternatives

Let us look at an example of a request to choose among alternatives:

```
Choose the routine you want:

List project titles
Delete a project
Modify a project title
Resequence projects
Quit

Type a letter (l, d, m, r, or q), then press ENTER.
```

The message has a headline that gives the significance of the list as routines from which to choose. The message ends with a recapitulation of what is to be done; it translates "choose the routine" into precise how-to instructions. (Note that the instructions are *not* "Enter a letter"; "enter" is our jargon for "press the ENTER key." This may confuse the novice.) Alternative Q is a "none of the above," which efficiently lets people reject all of the "positive" alternatives.

This message leaves to the context and to inference the answers to "What is being done?" and "Why are people being given a choice?" And we might say it leaves "to faith" the questions about consequences. (The consequences do not seem to be harmful. However, users might wonder whether there is any backing out from the choice to delete, or modify, or resequence. Of course, there should be.)

Requests for missing information

The program may not understand the input well enough to ask for a go-ahead, such as

Should I send the report to Jones?

or to give a choice among alternatives, such as

Do I send the report to Jones, Smith, or Williams?

Yet it may be able to describe the missing information it needs:

To whom should I send the report?

When the program asks for missing information, people need to know

- What is being done, or has already been done?
- What information are people being asked to give?
- What will happen if they give it, and what will happen if they do not?
- How do they give it?

Because people may be unable, or unwilling, to give information, they should always have the opportunity not to give it. The consequences of not giving it might be for the program to drop the transaction, to make do without the information, or to provide a default. Or, perhaps people can query the program for possible answers (in this case, the program might have requested a choice among alternatives in the first place).

Requests for correction or clarification of input

These are out-and-out "error messages." They should be issued only as a *last resort*. If a program can confidently correct or clarify the input, then no message of this type is needed. Or, with less confidence, the program might ask for a go-ahead. Or, it might give a choice among two or three alternative corrections or clarifications it can identify.

But sometimes the program does not understand the meaning of the input well enough to interpret it confidently. It cannot even confidently phrase its difficulty as a request for missing information. The program's only recourse is to discuss syntax or other conventions of how people give input. Programmers do not usually think of this as a last resort. They are comfortable with syntax; it is the way they view a large part of their professional world.

problems of syntax

But nonprogrammers are often uncomfortable with syntax, even the syntax of their native language. They are impatient, for example, with the explanation that in the sentence, "The audience applauded the pianist and I," the pronoun should be "me," not "I," because the verb takes a pronoun in the accusative case, not in the nominative.

If possible, phrase messages in terms of meaning rather than in terms of syntax. For example, we enter "display," thinking the computer will display the file we have just been working on. Here the program asks for correction in terms of syntax:

Missing operand in the DISPLAY command; add the name of the file.

The program talks of syntax even though it seems to "understand" what is going on and could have asked more naturally, in terms of meaning:

What file do you want to display?

This is not an error message! It is friendlier because it does not suggest that we failed to supply an operand.

fewer error messages

As much as anything, I am recommending that we issue fewer error messages, and instead issue messages that request a go-ahead, or give a choice among alternatives, or ask for missing information.

But when none of those choices is possible, and the program must ask for correction or clarification of input, people need to know

- What is being done, or has already been done?
- What is wrong with, or unclear about, the input?
- What will happen if people do not correct or clarify it?
- How do they correct or clarify it?

A particular problem in writing this type of message seems to be deciding how explicit to be. For example:

COMPARE is not processed, because record 101 does not have the same number of characters as record 247. Make the number of characters equal.

Probably anyone could infer the corrective action from the explanation, or the explanation from the corrective action. In fact, if the results of a comparison are usually displayed, people can probably infer from getting the message, with no display, that no comparison was done. Therefore, how about

The number of characters in record 101 is unequal to the number in record 247.

or

Make record 101 and record 247 have the same number of characters.

The first of these is negative. People process positive sentences significantly faster than they process negative sentences.[9] Thus, the second version may be the better choice.

Ideally, the program should identify the precise *one* thing wrong with, or unclear about, the input. Having identified one error, the program can issue a specific message.

When it cannot possibly identify only one error, the message should state each possibility (so long as there are no more than three) and clearly associate each possible error with its corresponding corrective action. (Why three? I cannot prove what the limit should be, but even five possible causes seem to me to be excessive in a message. When the number of possibilities exceeds three, it seems to me to be time not just to write the message differently, but to redesign the program.)

Summary of how to write messages

To determine what information to put in a message in a given situation, we first analyze what things people will need to know, then we determine whether they already know or can learn those things from the context. If not, the message must provide them, either explicitly or implicitly.

In deciding whether people can learn from the context or by inference from the message, we must respect novices as well as people familiar with our program. Novices do not get the same meanings from a context as experts do, nor do they infer as readily.

Playact to evaluate messages for usability before coding

After goals are set and messages written, we are ready to evaluate messages before we finish designing the program or system and begin coding. We would like to spot weaknesses in messages early enough to keep them out of the program in the first place.

Evaluating messages for usability is not the same as evaluating them for program function. Rather, it is to determine whether each message is relevant, specific, timely, and helpful. All of these qualities are relative to the people who receive a message and to the context in which they receive it. We cannot reliably evaluate messages for usability unless we consider the following: Who is the person receiving them—how experienced? how familiar with terminology? what reading ability? What is the person trying to do? How does the person understand the situation?

Before we start coding we are primarily interested in

- What messages are going to be sent? Are they the right messages? Will users need additional ones?
- Do messages give the right information? Are they specific enough?

To evaluate at this stage, we are going to have to rely a lot on playacting. We can playact entirely from a script, like actors sitting around and reading their parts at the first rehearsal. Or we can build a prototype of the program to make playacting a bit more realistic, as when the actors practice scenes on stage without having the script in their hands any longer.

playacting in use

Playacting from a script is not new—we have scenarios and walk-throughs. We can even have someone "play the program" and interact with someone else playing the user. This method is cheaper than building a prototype and can be very effective. We should do it, whatever else we do.

If we build a prototype or a driver, we can simulate program reactions to anticipated human input (so no one has to "play the program").

You should seriously consider finding people like those you expect will use the program and having them "use" the prototype. If real users might have big surprises for you, wouldn't you rather be surprised while you are designing the program than after you have worked hard to get the program working?

It may be easier to test a working program than to test a model, but it is harder to fix bad things in a working program than it is to fix them in a model.

Human factors departments have the facilities for testing prototypes. For one project I worked on, the human factors department in San Jose evaluated competitive prototypes. They hired representative people through a local employment agency and brought them in to participate in carefully controlled experiments.

Edit the messages for appropriate language

Editing messages is not the same as testing them for usability, although a sensitive, empathetic editor who is also a computer user will see many ways to improve messages to make the program easier to use. But an editor who is not intimately familiar with the use of a program cannot adequately judge, for example, whether message and context together meet the information needs of both novices and experts. To make that judgment, the editor must know a lot about the context. I have edited many messages in my time, and I know how frustrating it is to try to imagine contexts without adequate information.

editing points to check

But an editor can certainly check messages on the following points (in approximate order of importance) and correct messages as necessary:

- Are messages well written?
- Is the terminology accessible to the intended audience?
- Is standard conversational language used?
- Are messages consistent?
- Is punctuation standard?

Good writing. Of course, messages should be well-written—as anything should be that people will read. An editor typically finds numerous ways to shorten writing, but not for the sake of making it shorter, rather to make it clearer and easier to read. For that reason, the editor may very well want to lengthen messages that have been over-shortened by excessive use of such devices as abbreviation, contraction, symbols, and omission of small but useful words. These devices seldom improve communication; they are shorthand for the program or programmer's convenience.

Vocabulary that is familiar. The goal of editing messages for terminology is primarily to weed out technical terms and jargon that the users of the program probably will not understand (and should not be expected to). An editor is usually more sensitive to intrusive language than a programmer is, but frequently less sensitive than the real user. For this reason, editing cannot replace testing with representative users, who will have valuable comments about how messages affect them.

Standard conversational language. This is the language we use when we talk with a banker about a loan, or to a teacher about a child's progress in school. It is a respectful way of talking, though not stuffy. We say, or imply, "you" in referring to the person we are addressing (second person). We speak in the present tense predominantly. (We also say "please" and "thank you" and refer to ourselves as "I."[10])

Consistent messages. The messages a person sees in performing a set of related tasks should be consistent as a group. People expect certain things to be like other things they have experienced. Everyone on a project should agree on conventions beforehand and follow them in all messages they write, even though editing must be done later to ensure consistency. The order of parts in a message should generally be the same in all messages of that type. Exceptions must serve the higher goal of greater clarity or naturalness in particular messages. If messages are to have labels on each part ("status," "error," "action," or whatever), either *all* should have them or *none* should have them.

Standard punctuation. Standard punctuation is the punctuation used in business letters and professional journals. Statements end with a period, and questions end with a question mark. Dashes and semicolons are used sparingly. These practices are appropriate for messages.

Design the computer program or system to produce the messages

After you know your goals and have messages for the computer to produce, you will not get far with coding until you answer some technical questions:

- How will you provide multiple levels of detail?
- How will you let people ask for more information?
- How will you enable people to correct input, rather than resay it all?
- To what different destinations will you send related messages, and how will you sort them out?
- What parts of the program or system will send messages, and how will it know enough about the user to send the right ones?
- How will you construct messages that contain variable information?
- How can you monitor whether you are reaching your goals?

Your logic should practically design itself as you answer these questions. From my discussions with system programmers I can offer the following suggestions:

- Centralize and put outward the code that talks to people.
- Plan for usability improvement.

Put outward the code that talks to people

Program modules buried deep within a series of calls have little chance of issuing consistently good messages. These modules do not know what the human "out there" is trying to do. They simply serve the abstract needs of higher-level modules.

A person says to the computer,

```
send pmreport tom
```

A superior module asks an inferior: "Is this request syntactically correct?" The inferior module discovers that it is not, but this is all it knows. If it is responsible for giving the person the bad news, it can only say something like

```
Incorrect syntax. "tom" is an invalid operand.
```

Only the superior module, in touch with the person's situation, can interpret the abstract internal event ("syntax error") in a way appropriate to that situation. For example, the module can try variations that make sense:

```
"Send pmreport to m" makes sense, but no m exists.
"Send pmreport to tom" makes sense, and tom exists.
```

The module can perhaps issue the go-ahead message:

Do you mean, "send pmreport to tom"?

My point is that the part of the program in the best position to know the person (to know what is "going on outside") must do the communicating with the person. I do not mean that it should act as a mere switchboard, routing messages that originate elsewhere. This part of the program must relate events inside the program to events outside. It must make probable assumptions about what the person is doing and issue messages accordingly. It cannot do these things if it is buried inside. It must be outward.

Plan for usability improvement

One way to pinpoint weak human aspects of a program and its messages is to monitor what happens during interactions between people and program. One can create an audit trail of messages that indicate when people seem to be having a hard time. This log will show

- When the program asks for a go-ahead or gives a choice among alternatives because of incomplete or ambiguous input.
- When the program asks for missing information because of incomplete input.
- When the program's last resort is to ask for correction or clarification of input.

An auditor probably needs more information than just a record of what people and program say to each other. Some information (such as the time of occurrence) might be appended to each record saved, but other information might call for messages written especially for the auditor. These messages might include a profile of the person (how much experience?), a sketch of the system (its load, the mix of applications), and a summary of how much time the computer and the person are taking to interact.

Test the messages along with the running code

Testing a working program or system for usability is different from testing it for function. To test function, the tester "exercises the code." The most efficient way to do that is to understand the code and set up artificial test cases that anticipate what the code does. This is done from the point of view of the code.

usability testing

Usability testing has goals different from function testing, and must be done accordingly. Rather than exercise the code, the tester must either assume the user's point of view and use the program as a tool to do realistic tasks, or observe representative people using the program.

Testing the usability of a working program should be less imaginary than testing the design, even if testing is done by programmers who

playact at being representative users. The program is really there. It forces playacting to be more disciplined than it might be when the script is all on paper. Nevertheless, real representative users will give you a fairer test.

Consider automobile testing. The mechanics may not even take a car out of the garage for much of the testing, and when they drive it for "road-handling characteristics," they look for details that most drivers are never aware of. Mechanics, of course, have their own cars, like other people. They leave their overalls at the garage when they drive home. To some extent, mechanics can see a car as nonmechanics see it. But real nonmechanics (teachers, clerks, farmers, musicians) may be able to say more dependably whether or not a car is easy to drive. They do not have to overcome the disadvantage of knowing too much about automobiles.

Summary

I would now like to summarize briefly what has been discussed. I began by stating that one cannot produce good messages by just letting them grow like weeds during program development, then rewording them after the fact to make them read well. My experience and reflection have convinced me that, to create messages that work for their intended readers, we must do a number of things:

Set human goals for messages. We should first commit ourselves to certain goals. A major goal is to be tolerant of "user errors." We have to decide how much error-correction the program will do. By anticipating the kinds of mistakes humans will make, we can identify the messages needed to account for those mistakes, as well as for error-free operation.

Apply psychology in writing messages. We have an idea how people think and feel around computers. We ought to provide messages that match their expectations, that help them fit the pieces together the first time they read the messages, and that put them at ease. First decide what meaning to convey, then select the information to convey that meaning, and choose the language to present it.

Write messages for the audience and the situation. People want messages to suit their situation. To determine what information to put in a message in each situation, first analyze what things people will need to know, then determine whether they already know or can learn those things from the context. If not, the message must provide them, either explicitly or implicitly.

Playact to evaluate the messages for usability. We should evaluate messages for usability before coding begins to avoid big surprises later, when it is harder to repair things. We can either "play the

program" or have people who represent real users interact with a prototype while we observe them. This exercise in imagination is a must.

Edit the messages for appropriate language. Wording of messages is secondary to the point they make and to the information that is selected to make the point. But the language of messages must be edited for appropriate language: good writing, familiar terminology, standard conversational English, similarity among messages of the same type, standard punctuation.

Design the computer program or system to produce the messages. With messages written and evaluated, we can design the program or system to produce them and avoid the trap of compromising human requirements for the sake of program convenience. Perhaps the more challenging problems of computer science today lie not in inventing new function, but in making existing function easier to use.

Test the messages along with the running code. Messages should be evaluated for usability again after the program or system is working. People actually use it in realistic situations and observe what's what.

ACKNOWLEDGMENTS

Several people have helped me think about and develop this essay. One of them is Kevin Arnold, to whom I am grateful for suggesting that I ask, "What messages should a program send in the first place?" Jim Benjamin, Steve Uhlir, and Pat Wilson are system programmers who made many technical suggestions. Mr. Uhlir also gave me excellent editorial advice. The following people made useful comments on an early version of the essay: Terry Allard, Dudley Dinshaw, George Heigho, Phyllis Kaiser, Kris Malnarick, Edward Ort, James Overholt, Flo Pessin, Carol Schira, Bill Sprague, Doris Stoessel, Jim Vreeland, and Frank Waters.

CITED REFERENCES AND NOTE

1. T. Gilb and G. Weinberg, *Humanized Input: Techniques for Reliable Keyed Input*, Winthrop Publishers, Cambridge, MA (1977).
2. P. J. Hayes, J. E. Ball, and R. Reddy, "Breaking the man-machine communication barrier," *Computer* **14,** No. 3, 19–30 (March 1981).
3. Gilb and Weinberg, *op cit.*
4. M. Dean, "Using experimental psychology in technical writing," *Proceedings, 28th International Technical Communication Conference*, Society for Technical Communication, Pittsburgh, PA (May 1981), pp. E14–17. Also available as a Technical Report, TR 03.133, IBM Corporation, Santa Teresa Laboratory, San Jose, CA 95150 (March 1981).
5. E. D. Hirsch, Jr., *The Philosophy of Composition*, The University of Chicago Press, Chicago and London (1977). I discovered this fascinating book after I wrote the paper cited above. Hirsch helped me understand better how we use our short-term memory (and our long-term memory) when we read. His chapter on "The Psychological Bases of Readability" interprets some of the findings of G. A. Miller, published in *The Psychology of Communication*, Penguin, Baltimore (1967).

6. L. S. Chavarria, "Improving the friendliness of technical manuals," *Proceedings, 29th International Technical Communication Conference*, Society for Technical Communication, Boston (May 1982), pp. W26–28.
7. Gilb and Weinberg, *op cit.*
8. J. Martin, *Design of Man-Computer Dialogues*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1973).
9. From a workshop by W. E. Moody, "How Humans Read and Understand: Psychology Theory and Writing," at the Society for Technical Communication's 28th International Technical Communication Conference, Pittsburgh, PA (May 1981).
10. When I attended a usability symposium in 1981, I was told that Professor A. Chapanis, Director of the Communications Research Laboratory at The Johns Hopkins University, holds that people appreciate "please" and "thank you" in computer messages. "I" may smack too much of anthropomorphism.

The author is located at the IBM General Products Division, Santa Teresa Laboratory, P.O. Box 50020, San Jose, CA 95150.

Reprint Order No. G321-5175.