About Face 3

The Essentials of Interaction Design

Alan Cooper, Robert Reimann, and Dave Cronin

BICENTENNIAL 1807 WILEY 2007 BICENTENNIAL

Wiley Publishing, Inc.

About Face 3: The Essentials of Interaction Design
Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com
Copyright © 2007 Alan Cooper
Published by Wiley Publishing, Inc., Indianapolis, Indiana
Published simultaneously in Canada
ISBN: 978-0-470-08411-3
Manufactured in the United States of America
10 9 8 7 6 5 4

Library of Congress Cataloging-in-Publication Data:
Cooper, Alan, 1952-
About face 3 : the essentials of interaction design / Alan Cooper, Robert Reimann, and Dave Cronin.
p. cm.
Includes bibliographical references.
ISBN 978-0-470-08411-3 (pbk.)
1. User interfaces (Computer systems) 2. Human-computer interaction. I. Reimann, Robert. II. Cronin, Dave, 1972- III. Title. IV. Title: About face three.
QA76.9.U83C6596 2007
005.4'38--dc22

2007004977

12

Designing Good Behavior

As we briefly discussed in Chapter 10, research performed by two Stanford sociologists, Clifford Nass and Byron Reeves, suggests that humans seem to have instincts that tell them how to behave around other sentient beings. As soon as an object exhibits sufficient levels of interactivity — such as that found in your average software application — these instincts are activated. Our reaction to software as sentient is both unconscious and unavoidable.

The implication of this research is profound: If we want users to like our products, we should design them to behave in the same manner as a likeable person. If we want users to be productive with our software, we should design it to behave like a supportive human colleague. To this end, it's useful to consider the appropriate working relationship between human beings and computers.

DESIGN principle — The computer does the work and the person does the thinking.

The ideal division of labor in the computer age is very clear: The computer should do the work, and the person should do the thinking. Science fiction writers and computer scientists tantalize us with visions of artificial intelligence: computers that think for themselves. However, humans don't really need much help in the thinking department — our ability to identify patterns and solve complex problems creatively is unmatched in the world of silicon. We *do* need a lot of help with the work of information management — activities like accessing, analyzing, organizing, and visualizing information, but the actual decisions made from that information are best made by us — the "wetware."

Designing Considerate Products

Nass and Reeves suggest that software should be *polite*, but we prefer the term *considerate*. Although politeness could be construed as a matter of manners and protocol — saying "please" and "thank you," but doing little else helpful — being truly *considerate* means being concerned with the needs of others. Above and beyond performing basic functions, considerate software has the goals and needs of its users as a concern.

If an interactive product is stingy with information, obscures its processes, forces users to hunt around for common functions, and is quick to blame people for its own failings, users are sure to have an unpleasant and unproductive experience. This will happen regardless of how polite, cute, visually metaphoric, anthropomorphic, or full of interesting content the software is.

On the other hand, interactions that are respectful, generous, and helpful will go a long way toward creating a positive experience for people using your product.

DESIGN principle

Software should behave like a considerate human being.

Commonly, interactive products irritate us because they aren't considerate, not because they lack features. Building considerate products is not necessarily substantially more difficult than building rude or inconsiderate products. It simply requires that you envision interactions that emulate the qualities of a sensitive and caring person. None of these characteristics is at odds with more pragmatic goals of functional data processing (which lies at the core of all silicon-enabled products). In fact, behaving more humanely can be the most pragmatic goal of all, and if orchestrated correctly, this kind of dialogue with users can actually contribute to effective functional execution of software.

Humans have many wonderful characteristics that make them considerate, and some of these can be emulated to a greater or lesser degree by interactive products.

We think the following describe some of the most important characteristics of considerate interactive products (and humans):

- Take an interest
- Are deferential
- Are forthcoming
- Use common sense
- Anticipate people's needs
- Are conscientious
- Don't burden you with their personal problems
- Keep you informed
- Are perceptive
- Are self-confident
- Don't ask a lot of questions
- Take responsibility
- Know when to bend the rules

We'll now discuss these characteristics in detail.

Considerate products take an interest

A considerate friend wants to know more about you. He remembers your likes and dislikes so that he can please you in the future. Everyone appreciates being treated according to his or her own personal tastes.

Most software, on the other hand, doesn't know or care who is using it. Little, if any, of the *personal* software on our *personal* computers seems to remember anything *personal* about us, in spite of the fact that we use it constantly, repetitively, and exclusively. A good example of this behavior is the way that browsers such as Firefox and Microsoft Internet Explorer remember information that users routinely enter into forms on Web sites, such as a shipping address or username.

Software should work hard to remember our habits and, particularly, everything that we say to it. From the perspective of the programmer writing an application, it can be tempting to think about gathering a bit of information from a person as similar to gathering a bit of information from a database — every time the information is needed, the product asks the user for it. The application then discards that tidbit, assuming that it might change and that it can merely ask for it again if necessary. Not only are digital products better suited to recording things in memory than humans are, but our products also show they are *inconsiderate* when they forget. Remembering the actions and preferences of humans is one of the best ways to create a positive experience with a software-enabled product. We'll discuss the topic of memory in detail later in this chapter.

Considerate products are deferential

A good service provider defers to her client. She understands the person she is serving is the boss. When a restaurant host shows us to a table in a restaurant, we consider his choice of table to be a suggestion, not an order. If we politely request another table in an otherwise empty restaurant, we expect to be accommodated. If the host refuses, we are likely to choose a different restaurant where *our* desires take precedence over the host's.

Inconsiderate products supervise and pass judgment on human actions. Software is within its rights to express its *opinion* that we are making a mistake, but it is presumptuous for it to judge or limit our actions. Software can *suggest* that we not "submit" our entry until we've typed in our telephone number, and should explain the consequences if we do so, but if we wish to "submit" without the number, we expect the software to do as it is told. The very word *submit* and the concept it stands for are a reversal of the deferential relationship we should expect out of interactive products. Software should submit to users, and any application that proffers a "submit" button is being rude, as well as potentially oblique and confusing.

Considerate products are forthcoming

If you ask a good shop clerk for help locating an item, he will not only answer the question, but also volunteer useful collateral information; for example, the fact that a more expensive, higher-quality item than the one you requested is currently on sale for a similar price.

Most software doesn't attempt to provide related information. Instead, it narrowly answers the precise questions we ask it, and is typically not forthcoming about other information even if it is clearly related to our goals. When we tell our word processor to print a document, it doesn't tell us when the paper supply is low, or when 40 other documents are queued up before us, or when another nearby printer is free. A helpful human would.

Figuring out the right way to offer potentially useful information can require a delicate touch. Microsoft's "Clippy" is almost universally despised for his smarty-pants comments like "It looks like you're typing a letter, can I help?" While we applaud his sentiment, we wish he weren't so obtrusive and could take a hint when it's clear we don't want his help. After all, a good waiter doesn't ask you if you want more water. He just refills your glass when it's empty, and he knows better than to snoop around when it's clear that you're in the middle of an intimate moment.

Considerate products use common sense

Offering inappropriate functions in inappropriate places is a hallmark of poorly designed interactive products. Many interactive products put controls for constantly used functions directly adjacent to never-used controls. You can easily find menus offering simple, harmless functions adjacent to irreversible ejector-seat-lever expert functions. It's like seating you at a dining table right next to an open grill.

Horror stories also abound of customers offended by computer systems that repeatedly sent them checks for $0.00 or bills for $957,142,039.58. One would think that the system might alert a human in the Accounts Receivable or Payable departments when an event like this happens, especially more than once, but common sense remains a rarity in most information systems.

Considerate products anticipate human needs

A human assistant knows that you will require a hotel room when you travel to another city, even when you don't ask explicitly. She knows the kind of room you like and reserves one without any request on your part. She anticipates your needs.

A Web browser spends most of its time idling while we peruse Web pages. It could easily anticipate our needs and prepare for them while we are reading. It could use that idle time to preload all the links that are visible. Chances are good that we will soon ask the browser to examine one or more of those links. It is easy to abort an unwanted request, but it is always time-consuming to wait for a request to be filled. We'll discuss more ways for software to use idle time to our advantage towards the end of this chapter.

Considerate products are conscientious

A conscientious person has a larger perspective on 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 larger *goal*: cleaning up the kitchen. A conscientious person, when drafting a report, also puts a handsome cover page on it and makes enough photocopies for the entire department.

Here's an example: If we hand our imaginary assistant, 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 widgets that were delivered to MicroBlitz four months ago. The new folder, on the other hand, is for 32 sprockets slated for production and delivery in the next quarter. Conscientious Rodney changes the name on the old folder to read MicroBlitz Widget Contract, 7/03 and then changes the name of the new folder to read MicroBlitz Sprocket Contract, 11/03. This type of initiative is why we think Rodney is conscientious.

Our former imaginary assistant, Elliot, was a complete idiot. He was not conscientious at all, and if he were placed in the same situation he would have dumped the new MicroBlitz Contract folder next to the old MicroBlitz Contract folder without a second thought. Sure, he got it filed safely away, but he could have done a better job that would have improved our ability to find the right contract in the future. That's why Elliot isn't our imaginary assistant anymore.

If we rely on a word processor to draft the new sprocket contract and then try to save it in the MicroBlitz directory, the application offers the choice of either overwriting and destroying the old widget contract or not saving it at all. The application not only isn't as capable as Rodney, it isn't even as capable as Elliot. The software is dumb enough to make an assumption that because two folders have the same name, I meant to throw the old one away.

The application should, at the very least, mark the two files with different dates and save them. Even if the application refuses to take this "drastic" action unilaterally, it could at least show us the old file (letting us rename *that* one) before saving the new one. There are numerous actions that the application can take that would be more conscientious.

Considerate products don't burden you with their personal problems

At a service desk, the agent is expected to keep mum about her problems and to show a reasonable interest in yours. It might not be fair to be so one-sided, but that's the nature of the service business. An interactive product, too, should keep quiet about its problems and show interest in the people who use it. Because computers don't have egos or tender sensibilities, they should be perfect in this role, but they typically behave the opposite way.

Software whines at us with error messages, interrupts us with confirmation dialog boxes, and brags to us with unnecessary notifiers (Document Successfully Saved! How nice for you, Mr. Software: Do you ever *unsuccessfully* save?). We aren't interested in the application's crisis of confidence about whether or not to purge its Recycle Bin. We don't want to hear its whining about not being sure where to put a file on disk. We don't need to see information about the computer's data transfer rates and its loading sequence, any more than we need information about the customer service agent's unhappy love affair. Not only should software keep quiet about its problems, but it should also have the intelligence, confidence, and authority to fix its problems on its own. We discuss this subject in more detail in Chapter 25.

Considerate products keep us informed

Although we don't want our software pestering us incessantly with its little fears and triumphs, we do want to be kept informed about the things that matter to *us*. We don't want our local bartender to grouse to us about his recent divorce, but we appreciate it when he posts his prices in plain sight and when he writes what time the pregame party begins on his chalkboard, along with who's playing and the current Vegas spread. Nobody is interrupting us to tell us this information: It's there in plain view whenever we need it. Software, similarly, can provide us with this kind of rich modeless feedback about what is going on. Again, we discuss how in Chapter 25.

Considerate products are perceptive

Most of our existing software is not very perceptive. 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, 20 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, idling away billions of wasted instructions. It is not being perceptive. 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 your life, but maybe you'll want to get them for the rest of the week. Perceptive software observes what users are doing and uses those observations to offer relevant information.

Products should also watch our preferences and remember them without being asked explicitly to do so. If we always maximize an application to use the entire available screen, the application should get the idea after a few sessions and always launch in that configuration. The same goes for placement of palettes, default tools, frequently used templates, and other useful settings.

Considerate products are self-confident

Interactive products should stand by their convictions. If we tell the computer to discard a file, it shouldn't ask, "Are you sure?" Of course we're sure; otherwise, we wouldn't have asked. It shouldn't second-guess us or itself.

On the other hand, if the computer has any suspicion that we might be wrong (which is always), it should anticipate our changing our minds by being prepared to undelete the file upon our request.

How often have you clicked the Print button and then gone to get a cup of coffee, only to return to find a fearful dialog box quivering in the middle of the screen asking, "Are you sure you want to print?" This insecurity is infuriating and the antithesis of considerate human behavior.

Considerate products don't ask a lot of questions

As discussed in Chapter 10, inconsiderate products ask lots of annoying questions. Excessive choices quickly stop being a benefit and become an ordeal.

Choices can be offered in different ways. They can be offered in the way that we window shop. We peer in the window at our leisure, considering, choosing, or ignoring the goods offered to us — no questions asked. Alternatively, choices can be forced on us like an interrogation by a customs officer at a border crossing: *"Do you have anything to declare?"* We don't know the consequences of the question. Will we be searched or not? Software should never put users through this kind of intimidation.

Considerate products fail gracefully

When a friend of yours makes a serious faux pas, he tries to make amends later and undo what damage can be undone. When an application discovers a fatal problem, it has the choice of taking the time and effort to prepare for its failure without hurting the user, or it can simply crash and burn.

Many applications are filled with data and settings. When they crash, that information is often just discarded. The user is left holding the bag. For example, say an application is computing merrily along, downloading your e-mail from a server when it runs out of memory at some procedure buried deep in the internals of the application. The application, like most desktop software, issues a message that says, in effect, "You are completely hosed," and terminates immediately after you click OK. You restart the application, or sometimes the whole computer, only to find that the application lost your e-mail and, when you interrogate the server, you find that it has also erased your mail because the mail was already handed over to your application. This is not what we should expect of good software.

In our e-mail example, the application accepted e-mail from the server — which then erased its copy — but didn't ensure that the e-mail was properly recorded locally. If the e-mail application 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.

Some well-designed software products, such as Ableton Live, a brilliant music performance tool, rely upon the Undo cache to recover from crashes. This is a great example of how products can easily keep track of user behavior, so if some situation causes problems, it is easy to extricate oneself from that situation.

Even when applications don't crash, inconsiderate behavior is rife, particularly on the Web. Users often need to enter detailed information into a set of forms on a page. After filling in 10 or 11 fields, a user might click the Submit button, and, due to some mistake or omission on his part, have the site reject his input and tell him to correct it. The user then clicks the back arrow to return to the page, and lo, the 10 valid entries were inconsiderately discarded along with the single invalid one. Remember Mr. Jones, that incredibly mean geography teacher in junior high school who ripped up your entire report on South America and threw it away because you wrote using a pencil instead of an ink pen? Don't you hate geography to this day? Don't create products like Mr. Jones!

Considerate products know when to bend the rules

When manual information-processing systems are translated into computerized systems, something is lost in the process. Although an automated order-entry system can handle millions more orders than a human clerk can, the human clerk has the ability to *work the system* in a way most automated systems ignore. There is almost never a way to jigger the functioning to give or take slight advantages in an automated system.

In a manual system, when the clerk's friend from the sales force calls on the phone and explains that getting the order processed speedily means additional business, the clerk can expedite that one order. When another order comes in with some critical information missing, the clerk can go ahead and process it, remembering to acquire and record the information later. This flexibility is usually absent from automated systems.

In most computerized systems, there are only two states: nonexistence or full-compliance. No intermediate states are recognized or accepted. In any manual system, there is an important but paradoxical state — unspoken, undocumented, but widely relied upon — of **suspense**, wherein a transaction can be accepted although still not being fully processed. The human operator creates that state in his head or on his desk or in his back pocket.

For example, a digital system needs both customer and order information before it can post an invoice. Whereas the human clerk can go ahead and post an order in advance of detailed customer information, the computerized system will reject the transaction, unwilling to allow the invoice to be entered without it.

The characteristic of manual systems that lets humans perform actions out of sequence or before prerequisites are satisfied is called **fudgeability**. It is one of the first casualties when systems are computerized, and its absence is a key contributor to the inhumanity of digital systems. It is a natural result of the implementation model. Programmers don't see any reason to create intermediate states because the computer has no need for them. Yet there are strong human needs to be able to bend the system slightly.

One of the benefits of fudgeable systems is the reduction of mistakes. By allowing many small temporary mistakes into the system and entrusting humans to correct them before they cause problems downstream, we can avoid much bigger, more permanent mistakes. Paradoxically, most of the hard-edged rules enforced by computer systems are imposed to prevent just such mistakes. These inflexible rules cast the human and the software as adversaries, and because the human is prevented from fudging to prevent big mistakes, he soon stops caring about protecting the software from really colossal problems. When inflexible rules are imposed on flexible humans, both sides lose. It is invariably bad for business to prevent humans from doing what they want, and the computer system usually ends up having to digest invalid data anyway.

In the real world, both missing information and extra information that doesn't fit into a standard field are important tools 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 can only be completed if the termination date is 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 the time. Considerate products need to realize and embrace this fact.

Considerate products take responsibility

Too many interactive products take the attitude: "It isn't my responsibility." When they pass a job along to some hardware device, they wash their hands of the action, leaving the stupid hardware to finish up. Any user can see that the software isn't being considerate or conscientious, that the software isn't shouldering its part of the burden for helping the user become more effective.

In a typical print operation, for example, an application 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 clicks the Cancel button just as the first page emerges from the printer. The application 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 application 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 application smugly tells the user that the function was canceled. The application 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, he clicked the Cancel button, and then the stupid application continued printing for 15 pages even though he acted in plenty of time to stop it. It even acknowledged his Cancel command. As he throws the 15 wasted sheets of paper in the trash, he growls at the stupid application.

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 was built to be too indolent to use it.

Designing Smart Products

In addition to being considerate, helpful products and people must also be **smart.** Thanks to science fiction writers and futurists, there is some confusion about what it means for an interactive product to be smart. Some naive observers think that smart software is actually capable of behaving intelligently.

While this would certainly be nice, the fact of the matter is that our silicon-enabled tools are still a ways away from delivering on that dream. A more useful understanding of the term (if you're trying to ship a product this decade) is that these products are capable of working hard even when conditions are difficult and even when users aren't busy. Regardless of our dreams of thinking computers, there is a much greater and more immediate opportunity to get our computers to work harder. The remainder of this chapter discusses some of the most important ways that software can work a bit harder to serve humans better.

Putting the idle cycles to work

Because every instruction in every application 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, ensuring snappy performance for users. What we often forget, however, is that as soon as the CPU has hurriedly finished all its work, it waits idle, 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 needs to stop.

In our current computing systems, users need to remember too many things, such as the names they give to files and the precise location of those files in the file system. If a user wants to find that spreadsheet with the quarterly projections on it again, he must either remember its name or go browsing. Meanwhile, the processor just sits there, wasting billions of cycles.

Most current software also takes no notice of context. When a user is struggling with a particularly difficult spreadsheet on a tight deadline, for example, the application offers precisely as much help as it offers when he is noodling with numbers in his spare time. Software can no longer, in good conscience, waste so much idle time while users work. It is time for our computers to begin to shoulder more of the burden of work in our day-to-day activities.

Most users in normal situations can't do anything in less than a few seconds. That is enough time for a typical desktop computer to execute at least a *billion* 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 been: "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 application's assumptions are wrong; it has enough spare power to make several assumptions and discard the results of the bad ones when the user finally makes his choice.

With Windows and Mac OS X's preemptive, threaded multitasking and multicore, multichip computers, you can perform extra work in the background without significantly affecting the performance most users see. The application 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 application will have time to scan the whole disk. The user won't even notice. This is precisely the kind of behavior that makes Mac OS X's Spotlight search capabilities vastly superior to that in those windows. Search results are almost instantaneous because the operating system takes advantage of downtime to index the hard drive.

Every time an application puts up a modal 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? The application could, for example, offer the previous choice as a suggestion for this time.

We need a new, more proactive way of thinking about how software can help people reach their goals and complete their tasks.

Smart products have a memory

When you think about it, it's pretty obvious that for a person to perceive an interactive product as considerate and smart, that product must have some knowledge about the person and be capable of learning from their behavior. Looking through the characteristics of considerate products presented earlier reinforces this fact: For a product to be truly helpful and considerate it must *remember* important things about the people interacting with it.

Quite often, software is difficult to use because it operates according to rational, logical assumptions that, unfortunately, are very wrong. Programmers and designers often assume that the behavior of users is random and unpredictable, and that users must be continually interrogated to determine the proper course of action. Although human behavior certainly isn't deterministic like that of a digital computer, it is rarely random, and asking silly questions is predictably frustrating for users.

If your application, Web site, or device could predict what a user is going to do next, couldn't it provide a better interaction? If your application could know which selections the user will make in a particular dialog box or form, couldn't that part of the interface be skipped? Wouldn't you consider advance knowledge of what actions your users take to be an awesome secret weapon of interface design?

Well, you *can* predict what your users will do. You *can* build a sixth sense into your application that will tell it with uncanny accuracy exactly what the user will do next! All those billions of wasted processor cycles can be put to great use: All you need to do is give your interface a memory.

When we use the term **memory** in this context, we don't mean RAM, but rather a facility for tracking and responding to user actions over multiple sessions. If your application simply remembers what the user did the last several times (and how), it can use that as a guide to how it should behave the next time.

If we enable our products with an awareness of user behavior, a memory, and the flexibility to present information and functionality based upon previous user actions, we can realize great advantages in user efficiency and satisfaction. We would all like to have an intelligent and self-motivated assistant who shows initiative, drive, good judgment, and a keen memory. A product that makes effective use of its memory is more like that self-motivated assistant, remembering helpful information and personal preferences without needing to ask. Simple things can make a big difference: the difference between a product your users tolerate and one that they *love*. The next time you find your application asking your users a question, make it ask itself one instead.

You might think that bothering with a memory isn't necessary; it's easier to just ask the user each time. Many programmers are quick to pop up a dialog box to request any information that isn't lying conveniently around. But as we discussed in Chapter 10, *people don't like to be asked questions.* Continually interrogating users is not only a form of excise, but also, from a psychological perspective, it is a subtle way of expressing doubt about their authority.

Most software is forgetful, remembering little or nothing from execution to execution. If our applications *are* smart enough to retain any information during and between uses, it is usually information that makes the job easier for the *programmer* and not for the user. The application willingly discards information about the way it was used, how it was changed, where it was used, what data it processed, who used it, and whether and how frequently the various facilities of the application were used. Meanwhile, the application fills initialization files with driver names, port assignments, and other details that ease the programmer's burden. It is possible to use the exact same facilities to dramatically increase the smarts of your software from the perspective of the user.

Task coherence

Predicting what a user will do by remembering what he did last is based on the principle of **task coherence**: the idea that our goals and the way we achieve them (via tasks) is generally similar from day to day. This is not only true for tasks like brushing our teeth and eating our breakfasts, but it also describes how we use our word processors, e-mail applications, cell phones, and enterprise software.

When a consumer uses your product, there is a good chance that the functions he uses and the way he uses them will be very similar to what he did in previous uses of the product. He may even be working on the same documents, or at least the same types of documents, located in similar places. Sure, he won't be doing the exact same thing each time, but his tasks will likely be variants of a limited number of repeated patterns. With significant reliability, you can predict the behavior of your users by the simple expedient of remembering what they did the last several times they used the application. This allows you to greatly reduce the number of questions your application must ask the user.

Sally, for example, though she may use Excel in dramatically different ways than Kazu, will tend to use Excel the same way each time. Although Kazu likes 9-point Times Roman and Sally prefers 12-point Helvetica and uses that font and size with dependable regularity. It isn't really necessary for the application to ask Sally which font to use. A very reliable starting point would be 12-point Helvetica, every time.

Remembering choices and defaults

The way to determine what information the application should remember is with a simple rule: If it's worth the user entering, it's worth the application remembering.

DESIGN principle

If it's worth the user entering, it's worth the application remembering.

Any time your application finds itself with a choice, and especially when that choice is being offered to a user, the application should remember the information from run to run. Instead of choosing a hard-wired default, the application can use the previous setting as the default, and it will have a much better chance of giving a user what he wanted. Instead of asking a user to make a determination, the application should go ahead and make the same determination a user made last time, and let her change it if it was wrong. Any options users set should be remembered, so that the options remain in effect until manually changed. If a user ignores aspects of an application or turns them off, they should not be offered again. The user will seek them out when and if he is ready for them.

One of the most annoying characteristics of applications without memories is that they are so parsimonious with their assistance regarding files and disks. If there is one place where users need help, it's with files and disks. An application like Word remembers the last place a person looked for a file. Unfortunately, if she always puts her files in a directory called Letters, then edits a document template stored in the Template directory just one time, all her subsequent letters will be stored in the Template directory rather than in the Letters directory. So, the application must remember more than just the last place the files were accessed. It must remember the last place files *of each type* were accessed.

The position of windows should also be remembered, so if you maximized the document last time it should be maximized next time. If you positioned it next to another window, it is positioned the same way the next time without any instruction from the user. Microsoft Office applications now do a good job of this.

Remembering patterns

Users can benefit in several ways from a product with a good memory. Memory reduces excise, the useless effort that must be devoted to managing tools and not doing work. A significant portion of the total excise of an interface is in having to explain things to the application that it should already know. For example, in your word processor, you might often reverse-out text, making it white on black. To do this, you select some text and change the font color to white. Without altering the selection, you then set the background color to black. If the application paid enough attention, it would notice the fact that you requested two formatting steps without an intervening selection option. As far as you're concerned, this is effectively a single operation. Wouldn't it be nice if the application, upon seeing this unique pattern repeated several times, automatically created a new format style of this type — or better yet, created a new Reverse-Out toolbar control?

Most mainstream applications allow their users to set defaults, but this doesn't fit the bill as a memory would. Configuration of this kind is an onerous process for all but power users, and many users will never understand how to customize defaults to their liking.

Actions to remember

Everything that users do should be remembered. There is plenty of storage on our hard drives, and a memory for your application is a good investment of storage space. We tend to think that applications are wasteful because a big application might consume 200 MB of disk space. That is typical usage for an application, but not for user data. If your word processor saved 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 10 times every business day. There are approximately 200 workdays per year, so you run the application 2000 times a year. The net consumption is still only 2 MB, and that gives an exhaustive recounting of the entire year! This is probably not much more than the background image you put on your desktop.

File locations

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

Deduced information

Software should not simply remember these kinds of explicit facts, but should also remember useful information that can be deduced from these facts. For example, if the application 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, and 92. If the user calls up the file and changes 100 bytes, nothing would be out of the ordinary. But if the number of changed bytes suddenly shoots up to 5000, the application might suspect that something is amiss. Although there is a chance that the user has inadvertently done something about which he will be sorry, the probability of that is low, so it isn't right to bother him with a confirmation dialog. It is, however, very reasonable for the application to make sure to keep a milestone copy of the file before the 5000 bytes were changed, just in case. The application probably won't need to keep it beyond the next time the user accesses that file, because the user will likely spot any mistake that glaring immediately, and he would then demand an undo.

Multisession undo

Most applications discard their stack of undo actions when the user closes the document or the application. This is very shortsighted on the application's part. Instead, the application could write the undo stack to a file. When the user reopens the file, the application could reload its undo stack with the actions the user performed the last time the application was run — even if that was a week ago!

Past data entries

An application with a better memory can reduce the number of errors users make. This is simply because users have to enter less information. More of it will be entered automatically from the application's memory. In an invoicing application, for example, if the software enters the date, department number, and other standard fields from memory, the invoicing clerk has fewer opportunities to make typing errors in these fields.

If the application remembers what the user enters and uses that information for future reasonableness checks, the application can work to keep erroneous data from being entered. Contemporary Web browsers such as Internet Explorer and Firefox provide this facility: Named data entry fields remember what has been entered into them before, and allow users to pick those values from a combo box. For security-minded individuals, this feature can be turned off, but for the rest of us, it saves time and prevents errors.

Foreign application activities on application files

Applications might also leave a small thread running between invocations. This little application can keep an eye on the files it worked on. It can track where they go and who reads and writes to them. This information might be helpful to a user when he next runs the application. When he tries to open a particular file, the application can help him find it, even if it has been moved. The application can keep the user informed about what other functions were performed on his file, such as 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.

Applying memory to your applications

A remarkable thing happens to the software design process when developers accept the power of task coherence. Designers find that their thinking takes on a whole new quality. The normally unquestioned recourse of popping up a dialog box gets replaced with a more studied process, where the designer asks questions of much greater subtlety. Questions like: How *much* should the application remember? Which aspects should be remembered? Should the application remember more than just the last setting? What constitutes a change in pattern? Designers start to imagine situations like this: The user accepts the same date format 50 times in a row, and then manually enters a different format once. The next time the user enters a date, which format should the application use? The format used 50 times or the more recent one-time format? How many times must the new format be specified before it becomes the default? Just because there is ambiguity here, the application still shouldn't ask the user. It must use its initiative to make a reasonable decision. The user is free to override the application's decision if it is the wrong one.

The following sections explain some characteristic patterns in the ways people make choices that can help us resolve these more complex questions about task coherence.

Decision-set reduction

People tend to reduce an infinite set of choices down to a small, finite set of choices. Even when you don't do the exact same thing each time, you will tend to choose your actions from a small, repetitive set of options. People unconsciously perform this **decision-set reduction**, but software can take notice and act upon it.

For example, just because you went shopping at Safeway yesterday doesn't necessarily mean that you will be shopping at Safeway exclusively. However, the next time you need groceries, you will probably shop at Safeway again. Similarly, even though your favorite Chinese restaurant has 250 items on the menu, chances are that you will usually choose from your own personal subset of five or six favorites. When people drive to and from work, they usually choose from a small number of favorite routes, depending on traffic conditions. Computers, of course, can remember four or five things without breaking a sweat.

Although simply remembering the last action is better than not remembering anything, it can lead to a peculiar pathology if the decision set consists of precisely two elements. If, for example, you alternately read files from one directory and store them in another, each time the application offers you the last directory, it will be guaranteed to be wrong. The solution is to remember more than just one previous choice.

Decision-set reduction guides us to the idea that pieces of information the application must remember about the user's choices tend to come in groups. Instead of one right way, several options are all correct. The application should look for more subtle clues to differentiate which one of the small set is correct. For example, if you use a check-writing application to pay your bills, the application may very quickly learn that only two or three accounts are used regularly. But how can it determine from a given check which of the three accounts is the most likely to be appropriate? If the application remembered the payees and amounts on an account-by-account basis, that decision would be easy. Every time you pay the rent, it is the exact same amount! It's the same with a car payment. The amount paid to the electric company might vary from check to check, but it probably stays within 10 or 20% of the last check written to them. All this information can be used to help the application recognize what is going on, and use that information to help the user.

Preference thresholds

The decisions people make tend to fall into two primary categories: important and unimportant. Any given activity may involve hundreds of decisions, but only a few of them are important. All the rest are insignificant. Software interfaces can use this idea of **preference thresholds** to simplify tasks for users.

After you decide to buy that car, you don't really care who finances it as long as the terms are competitive. After you decide to buy groceries, the particular check-out aisle you select may not be important. After you decide to ride the Matterhorn at Disneyland, you don't really care which toboggan they seat you in.

Preference thresholds guide us in our user interface design by demonstrating that asking users for successively detailed decisions about a procedure is unnecessary. After a user asks to print, we don't have to ask him how many copies he wants or whether the image is landscape or portrait. We can make an assumption about these things the first time out, and then remember them for all subsequent invocations. If the user wants to change them, he can always request the Printer Options dialog box.

Using preference thresholds, we can easily track which facilities of the application each user likes to adjust and which are set once and ignored. With this knowledge, the application can offer choices where it has an expectation that a user will want to take control, not bothering him with decisions he won't care about.

Mostly right, most of the time

Task coherence predicts what users will do in the future with reasonable, but not absolute, certainty. If our application relies on this principle, it's natural to wonder about the uncertainty of our predictions. If we can reliably predict what the user will do 80% of the time, it means that 20% of the time we will be wrong. It might seem that the proper step to take here is to offer users a choice, but this means that they will be bothered by an unnecessary dialog 80% of the time. Rather than offering a choice, the application should go ahead and do what it thinks is most appropriate and allow users to override or undo it. If the undo facility is sufficiently easy to use and understand, users won't be bothered by it. After all, they will have to use undo only two times out of ten instead of having to deal with a redundant dialog box eight times out of ten. This is a much better deal for humans.