CH 8

# DIGITAL ETIQUETTE

In *The Media Equation* (Cambridge University Press, 1996), Stanford sociologists Clifford Nass and Byron Reeves make a compelling case that humans treat and respond to computers and other interactive products as if they were people. We should thus pay real attention to the “personality” projected by our digital products. Are they quietly competent and helpful, or do they whine, nag, badger, and make excuses?

Nass and Reeves’ research suggests that humans 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 digital 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 that much help in the thinking department. Our ability to identify patterns and solve complex problems creatively is currently 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-enabled products 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 that is 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.*

Building considerate products is not necessarily more difficult than building rude or inconsiderate products, but it requires that you envision interactions that emulate the qualities of a sensitive and caring person. 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 are some of the most important characteristics of considerate interactive products (and humans):

- Take an interest
- Are deferential
- Are forthcoming

- Use common sense
- Use discretion
- 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
- Fail gracefully
- Know when to bend the rules
- Take responsibility
- Help you avoid awkward mistakes

None of these characteristics, by the way, is at odds with more pragmatic goals of functional data processing (which lies at the core of all silicon-enabled products). Let's discuss them in more 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 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 how browsers such as Firefox and Microsoft Internet Explorer remember information that users routinely enter into forms on websites, such as a shipping address or username. Google Chrome even remembers these details across devices and sessions.

Software should work hard to remember our habits and, particularly, everything we tell it. From the perspective of a developer writing an application, it can be tempting to think about gathering a bit of information from a person as being 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 humans' actions and preferences 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 that 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 it should explain the consequences if we do so, but if we want to "submit" without the number, we expect the software to do as it is told. The very word *submit* is a reversal of the deferential relationship we should expect from interactive products. Software should submit to users. Any application that proffers a "submit" button is being rude, as well as oblique and confusing.

## Considerate products are forthcoming

If you ask a good retail sales associate for help locating an item, he will not only answer your question, but also volunteer useful collateral information. For example, he might tell you 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 typically is 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 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 Office Assistant "Clippy" was almost universally despised for his smarty-pants comments like "It looks like you're writing a letter. Would you like help?" While we applauded his sentiment, we wished he weren't so obtrusive and could take a hint when it was clear we didn't want his help. After all, a good waiter doesn't interrupt your conversation to ask you if you want more water. He just refills your glass when it's empty, and he knows better than to linger 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 right next to never-used controls. You can easily find menus offering simple, harmless functions adjacent to irreversible ejector-seat-lever expert functions. It's like being seated 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 use discretion

Generally speaking, we want our software to remember what we do and what we tell it. But there are some things our that software probably shouldn't remember unless we specifically direct it, such as credit card numbers, tax IDs, bank accounts, and passwords. Furthermore, it should help us protect this kind of private data by helping us choose secure passwords, and reporting any possible improprieties, such as accounts accessed from an unrecognized computer or location.

## Considerate products anticipate people's 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 later in 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, he checks the writing on the folder's tab—let's say it reads MicroBlitz Contract—and proceeds to find the correct place for it in the filing cabinet. Under M, he finds, to his surprise, a manila folder already there with the same name. Rodney notices the discrepancy and finds that the existing folder contains a contract for 17 widgets delivered four months ago. The new folder, on the other hand, is for 34 sprockets slated for production and delivery in the next quarter. Conscientious Rodney changes the name on the old folder to read MicroBlitz Widget Contract, 7/13 and then changes the name of the new folder to read MicroBlitz Sprocket Contract, 11/13. 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. 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 safely filed, 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 typical word processor to draft the new sprocket contract and then try to save it in the MicroBlitz folder, 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 assume that because two folders have the same name, we meant to throw away the old one.

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. The application could take numerous actions  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. App: Do you ever *unsuccessfully* save?). We aren't interested in the application's crisis of confidence about whether to purge its Recycle Bin. We don't want to hear its whining about being unsure 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 15.

## Considerate products keep you 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 interrupts 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 15.

## 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. For example, if you ask the inventory query system 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

Inconsiderate products ask lots of annoying questions. Excessive choices, especially in the form of questions, quickly stop being a benefit and instead become an ordeal.

Asking questions is quite different from providing choices. When browsing on your own in a store, you are presented with *choices*. When going to a job interview, you are asked *questions*. Which is the more pleasurable experience? Part of the reason why is that individual asking the questions is understood to be in a position superior to the individual being asked. Those with authority ask questions; subordinates respond. When software asks questions rather than offering choices, users feel disempowered.

Beyond the power dynamics issues, questions also tend to make people feel badgered and harassed. *Would you like soup or salad?* Salad. *Would you like cabbage or spinach?* Spinach. *Would you like French, Thousand Island, or Italian?* French. *Would you like lite or regular?* Stop! Please, just bring me the soup instead! *Would you like corn chowder or chicken noodle?*

Users *really* don't like to be asked questions by products, especially since most of the questions are stupid or unnecessary. Asking questions tells users that products are:

- Ignorant
- Forgetful
- Weak
- Fretful
- Lacking initiative
- Overly demanding

These are all qualities that we typically dislike in people. Why should we desire them in our products? The application is not asking us our opinion out of intellectual curiosity or a desire to make conversation, the way a friend might over dinner. Rather, it is behaving ignorantly while also representing itself as having false authority. The application isn't interested in our opinions; it requires information—often information it didn't really need to ask us in the first place.

Many ATMs continually ask users what language they prefer: "Spanish, English, or Chinese?" This answer is unlikely to change after a person's first use. Interactive products that ask fewer questions, provide choices without asking questions, and remember information they have already learned appear smarter to users, as well as more polite and considerate.

## Considerate products fail gracefully

When your friend commits a serious faux pas, he tries to make amends and undo the damage. When an application discovers a fatal problem, it can take 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 still often 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 application's internals. The application, like most desktop software, issues a message that says, in effect, "You are hosed," and it 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. When you interrogate the server, you find that it has also erased your e-mail because the e-mail was already handed over to your application. This is not what we should expect of good software.

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

Even when applications don't crash, inconsiderate behavior is rife, particularly on the web. Users often need to enter 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 your incredibly mean junior high geography teacher  who ripped up your report on South America because you wrote it in pencil instead of ink? And as a result, you hate geography to this day? Don't create products that act like that!

## 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 can *work the system* in a way that 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 tells him that getting a particular 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 still can process it, remembering to acquire and record the information later. This flexibility usually is absent from automated systems.

Most computerized systems have only two states: nonexistence and full compliance. No intermediate states are recognized or accepted. Every manual system has an important but paradoxical state—unspoken, undocumented, but widely relied upon—**suspense**. In this state a transaction can be accepted even though it is not fully processed. The human operator creates that state in his head or on his desk or in his back pocket.

For example, suppose 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 before having detailed customer information, the computerized system rejects 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. Developers don't always see a 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 reducing the number 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 make the human and the software adversaries. Because the human is prevented from fudging to prevent big mistakes, he soon stops caring about protecting the software from colossal problems. When inflexible rules are imposed on flexible humans, both sides lose. It is bad for business to prevent humans from doing things the way 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. For example, suppose a transaction can be completed only 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 that "It isn't my responsibility." When they pass along a job to some hardware device, they wash their hands of the action, leaving the stupid hardware to finish. Any user can see that the software isn't being considerate or conscientious, that the software isn't shouldering its part of the burden of helping the user become more effective.

In a typical print operation, for example, an application begins sending a 20-page report to the printer and simultaneously displays 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 had already sent 15 pages to the printer's buffer. The application cancels the last five pages, but the printer doesn't know 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 is unsympathetic 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 tray, he clicked the Cancel button, and then the stupid application continued printing for 15 pages even after it acknowledged his Cancel command.

Imagine what his experience would be if the application could properly communicate with the print driver. If the software were smart enough, the print job could easily have been abandoned before the second sheet of paper was wasted. The printer has a Cancel function—it's just that the software was built to be too lazy to use it.

### Considerate products help you avoid awkward mistakes

If a helpful human companion saw you about to do something that you would almost certainly regret afterwards—like shouting about your personal life in a room full of strangers, or sending an empty envelope in the mail to your boss—they might take you quietly aside and gently alert you to your mistake.

Digital products should similarly help you realize when you are, for example, about to inadvertently send a text to your entire list of contacts instead of the one friend you were intending to confide in, or are about to send an e-mail to the director of your department without the quarterly report you mentioned you were enclosing in the text of your message.

The intervention should not, however, be in the form of a standard modal error message box that stops the action and adds insult to injury, but rather through careful visual and textual feedback that lets you know that you are messaging a group rather than a person, or that you haven't enclosed any attachments even though you mentioned that you were.

For this latter situation, your e-mail app might even modelessly highlight the drop area for you to drag your attachment to, while at the same time giving you the option of just going ahead and sending the message sans attachments, in case the software got your intentions wrong.

Products that go the extra mile in looking out for users by helping them prevent embarrassing mistakes—and not berating them for it—will quickly earn their trust and devotion. All other things equal, considerate product design is one of the things, and perhaps even *the* thing, that distinguishes an only passable app from a truly great one.

## Designing Smart Products

In addition to being considerate, helpful products and people must 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 can actually behave intelligently.

While that might be nice, 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 can work 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.

## Smart products put idle cycles to work

Because every instruction in every application must pass single-file through a CPU, we tend to optimize our code for this needle's eye. Developers 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 effort 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, 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 struggles 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 shouldering 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 Apple's OS X's preemptive, threaded multitasking and multicore, multichip computers, the computer 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 the search until the next hiatus. Eventually, the user stops to think, and the application has time to scan the whole disk. The user won't even notice. This is precisely the kind of behavior that makes OS X's Spotlight search capabilities so good. 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 of the person and be capable of learning from his or her 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.

Developers and designers often assume that user behavior 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, so being asked silly questions is predictably frustrating for users.

Because of assumptions like this, 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 *developer*, not the user. The application willingly discards information about how it was used, how it was changed, where it was used, what data it processed, who used it, and whether and how frequently the application's various facilities were used. Meanwhile, the application fills initialization files with driver names, port assignments, and other details that ease the developer's burden. It is possible to use the exact same facilities to dramatically increase the software's smarts from the user's perspective.

If your application, website, or device could predict what a user will 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 on 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 they *love*. The next time you find your application asking your users a question, make it ask itself one instead.

## Smart products anticipate needs

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

When a consumer uses your product, there is a good chance that the functions he uses and how 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 your users' behavior 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.

For example, even though Sally may use Excel in dramatically different ways than she does PowerPoint or Word, she tends to use Excel the same way each time. 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 or to revert to the default. A reliable starting point would be 12-point Helvetica, every time.

Applications can also pay closer attention to clusters of actions. 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 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 smart behavior. Configuration of this kind is an onerous process for all but power users. Many users will never understand how to customize defaults to their liking.

## Smart products remember details

You can determine what information the application should remember with a simple rule: If it's worth it to the user to do it, it's worth it to the application to remember it.

*Everything* that users do should be remembered. Our hard drives have plenty of storage, 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 might be typical usage for an application, but not for most user data. If your word processor saved 1 KB of execution notes every time you ran it, it still wouldn't amount to much. Suppose you use your word processor 10 times every business day. There are approximately 250 workdays per year, so you run the application 2,500 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.

DESIGN PRINCIPLE

*If it's worth it to the user to do it, it's worth it to the application to remember it.*

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 hardwired default, the application can use the previous setting as the default, and it will have a much better chance of giving the user what he wants. Instead of asking the user to make a determination, the application should go ahead and make the same determination the user made last time, and let him change it if it is 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 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 folder called Letters, and then she edits a document template stored in the Templates folder just once, all her subsequent letters are saved in the Templates folder rather than in the Letters folder. 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; 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 do a good job of this.

## Remembering file locations

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

## Deducing information

Software should not simply remember these kinds of explicit facts; it should also remember useful information that can be deduced from these facts. For example, if the application remembers how many bytes changed in the file each time it is opened, it can help the user with some checks of reasonableness. Imagine that a file's changed-byte count is 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 5,000 (as a result, perhaps, of inadvertently deleting a page of content), the application might suspect that something is amiss. Although there is a chance that the user has inadvertently done something that will make him 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 keep a milestone copy of the file before the 5,000 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 glaring mistake and will then perform an Undo.

### Multi-session Undo

Most applications discard their stack of Undo actions when the user closes the document or 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. Many browsers offer this function, though few mobile or desktop applications do. 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 those 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 has worked on. It can track where the files 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 it was printed or e-mailed. Sure, this information might be unneeded, but the computer can easily spare the time, and only bits have to be thrown away, after all.

## Making smart products work

A remarkable thing happens to the software design process when the power of task coherence is recognized. 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 in which the designer asks much more subtle questions. How much should the application remember? Which aspects should it remember? 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? Even though ambiguity exists, 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 how people make choices. These can help us resolve these more-complex questions about task coherence.

## Decision-set reduction

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

For example, just because you went shopping at Safeway yesterday doesn't necessarily mean you shop at Safeway exclusively. However, the next time you need groceries, you probably will shop at Safeway again. Similarly, even though your favorite Chinese restaurant has 250 items on the menu, chances are you usually choose from your 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. For example, if you alternately read files from one folder and store them in another, each time the application offers you the last folder, it is 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 percent of the last check written. 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 checkout aisle you select may be unimportant. After you decide to ride the Matterhorn at Disneyland, you don't really care which bobsled 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 chooses 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 the 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 percent of the time, this means that 20 percent 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 box 80 percent 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.

## Designing Social Products

Most of the content created in modern software isn't made for the sole use of its author (task lists, reminders, and personal journals are perhaps the exceptions). Most content is passed on to other humans: Presentations to be heard, documents to be read, images to be seen, status updates to be liked; as well as content to be reviewed, application, and used in the next step of some business process. Even lines of application code that are "meant for machines," are still written in structures and using language that is meant to be readable by other developers.

Traditional, non-connected software left the sharing of this content up to the user, via documents and through software that was specifically built for the task, such as an e-mail client. But as software has grown more complex—and more respectful of the goals of its users—those sharing and collaboration features are built right in. Software is becoming less like a private office into which one disappears to produce work and later emerges, work-in-hand, but more like an open-plan office where collaborators and even consumers are just a shout away.

So far, this chapter has discussed how good software should be considerate to humans. When we communicate with other users via software-mediated mechanisms, the principle is the same, but must additionally take into account social norms and the expectations of the other users.

### Social products know the difference between social and market norms

Every group has its own set of rules to which its members adhere, but the two largest categories are *social norms* and *market norms*.

*Social norms* are the unspoken rules of reciprocity afforded to friends and family. They include things like lending a hand when in need, and expressing gratitude.

*Market norms* are a different set of unspoken rules afforded to people with whom one is doing business. They include things like assurances of fair price for quality goods and honesty in dealings.

These two concepts are very distinct from each other. Mistakenly using market norms in a social setting can be perceived as extremely rude. Imagine leaving money on the table after enjoying a good meal at a friend's house and then leaving. Mistakenly using social norms in a market setting can get you arrested. Imagine shaking the hand of the waiter at a restaurant and saying, "Thank you, that was delicious!" and leaving without paying.

If your software is to avoid being either rude or illegal, it must know the norms in which its users operate. This is quite often apparent given the domain, but more all-encompassing systems may toy with those boundaries. You are connected to friends through LinkedIn. There are companies with pages on Facebook. Software used internally at an organization operates on semi-social rules, since its members are helping each other do the organization's business. Dating software are actually markets where the transaction is agreeing to meet.

Software adhering to market norms should assure both parties in a transaction that the deal will be fair, often acting as the "matchmaker" and trustworthy escrow agent. Software adhering to social norms should help its users adhere to the subculture's rules and hierarchy in reciprocal, rewarding ways.

## Social software lets users present their best side

Representing users in a social interface poses challenges to the designer, that is, how to make those representations unique, recognizable, descriptive, and useful. The following strategies can help users represent themselves online.

### User identity

It's tempting to want to use names to represent a user's identity, but names aren't always unique enough, and they don't always lend themselves to compact representations.

It's also tempting to assign semi-random avatars, as Google Docs does with its colored animal icons (while typing this, the author has been joined by an adorable magenta panda). The use of semi-random but canned visuals lets designers keep user avatars in line with the software's look and feel, but this adds cognitive load, as the user must then remember the person behind the icon. To keep this load to a minimum, let the user on the other end provide a visual that works best for them, either by selecting an unclaimed icon-color combination themselves, or by uploading their own image.

Uploading images does introduce the risk that inappropriate content might be chosen, but within opt-in networks and accountable (non-anonymous) social software, social pressures should keep this in check. In any case, adding a ToolTip for a full name offers users a quick reminder about who is who, should they need it.

### Dynamic vs static user profiles

User profiles are a traditional way that a user can create an online presence for themselves, but not all users have the time or desire to fill out a bunch of static fields describing themselves. However, a user's social contributions to the network can be dynamically collected and presented in summary: music listened to, people followed or friended, updates added or liked, books or movies mentioned, links posted or shared, etc. Facebook's Timeline is an example of one way of presenting this kind of information. In social networks, actions speak louder than self-descriptions, and summarized top-rated or best-liked contributions can form a great supplement or even an alternative to static bios.

As with any part of a user's profile, a user should have complete control of who gets to see this information, and should be able to curate it and organize it as they see fit—though the default should also look good for those users who don't want to fuss with it.

## Social software permits easy collaboration

Collaboration is one of the most common reasons to make software social. A user may desire a second opinion, another pair of hands, or signoff by a colleague. Collaboration-centric social software can bake this type of functionality deeply into the interface.

For example, Microsoft Word lets people add review comments to documents, and others can add comments referencing earlier ones. It works after a fashion, but comments can unfortunately get jumbled and dissociated from their source.

Google Docs does Word one better by allowing collaborators to reply directly to comments, treating them almost as a threaded discussion, lets any collaborator "resolve" an issue with the click of a button, and supplies a place where old, resolved issues can be re-opened. The Google Docs model better fits the ways people interact to discuss and resolve questions. Designers should make sure that collaboration tools are apparent, usable, and fit their personas' collaboration needs and communication behaviors.

## Social products know when to shut the door

In productivity software that is only peripherally social, such as Google Docs, the socialness of the software should not overwhelm or distract from the primary task. Of course social software must manifest other users, but it should do so carefully, treating the attention of the user with the utmost respect. A blinking, beeping announcement that an already-invited user has come into my document is a surefire way to get me to find another alternative. Additionally, users should have access to a polite but firm way to "shut the door," suspending social interruption for a time that the user needs to focus to get a task done.

## Social products help networks grow organically

Networks of people grow and change over time as members join, connect, interact, gain standing, and leave. Social software must have ways for new members to discover the network, join, build a presence, learn the rules, begin to participate, and receive gentle reinforcements when the member transgresses against the subculture's norms. Intermediate members should have tools that let them build mastery, nurture newer members, and seek help from senior members. Senior members should have tools to manage the network. All members need ways to suspend participation for a time, or to elegantly bow out. And, as morbid as the thought may be, networks must elegantly handle when their members die.

A particularly difficult social problem is how to handle when one user wishes to connect to another user who does not agree to the connection. Per social norms, the reluctant user can come across as aloof or standoffish. When the new mailroom intern tries to connect via LinkedIn to the CEO of a large organization, the CEO doesn't want to connect, but also doesn't want to send a signal that that employee isn't valued. To let the eager user save face, the reluctant user needs to pass responsibility to a set of explicit rules for the community, to another user who is expressly charged as a gatekeeper, or to some system constraint.

## Social products respect the complexity of social circles

There is a psychosocial threshold to the size of networks that designers must keep in mind. Research from primatology has revealed that the size of the neocortex in a primate's brain has a direct correlation to the average size of its tribe. Tribes with a number of members smaller than this value risk having less security. Tribes with a number of members larger than this value become unstable.

This limit is called *Dunbar's number*, named for the anthropologist who first proposed it, Robin Dunbar. The obvious follow-up question to the primate observation is, of course, what is that value for humans? Given the size of our neocortex, the value is probably around 150. This is a generalized maximum number of social relationships that any one person can maintain fully. If your software permits networks larger than this value, there is a risk of instability unless there are either explicit rules and mechanisms that proscribe behavior, or a set of tools to manage the subsequent complexity.

Some software is slightly social, such as Google Docs, which allows one user to invite others to collaborate on individual documents. The social group here is strictly invite-only and strictly opt-in. Users may have a few levels of access, determined by the owner of the original document. These types of software leave complexity as an exercise for the user.

Other software may have more deliberate networks and rules around participation. Dropbox, for instance, is a file sharing software that allows users to define a network of who has access to a shared group of files. Users may invite others, even outside of the defined organization, to individual folders or files.

Explicitly social software can involve massively disparate circles of people. Facebook, with its estimated 1 billion users, can find its users connected to coworkers, nuclear family, extended family, friends, frenemies, neighbors, ex-neighbors, school mates, ex-school mates, acquaintances, hobbyist circles, employers, potential employers, employees, clients, potential clients, sweethearts, paramours, and suspicious rivals, to name a few. The rules and norms for each of these subgroups can be of critical importance to the user, as can keeping groups separate.

For example, your college friends may delight in photos of your night of irresponsible carousing, but only some of your family would, and your employer or clients would almost certainly not. Unfortunately the mechanism for managing these circles and specifying which content goes to which circle is hard to find and cumbersome to use, occasionally resulting in some very embarrassed Facebook users. The ability to change permissions on the fly and retract actions is critical in such contexts. Facebook's rival social network, Google Plus, provides its users with a much clearer mechanism for understanding and managing social circles, with animations and modern layouts that make it fun rather than a chore.

Other more explicit *communities of practice* may afford certain permissions to members depending on role and seniority. Wikipedia has a huge amount of readers, a smaller group that writes and edits pages, and an even smaller group that has the administrative ability to grant permissions.

The larger and more complex your software is, the more design attention you will need to pay to the complexity of the social network.

## Social products respect other users' privacy

Advertising-driven social media, like Facebook and Google Plus, have strong financial incentives to disrespect their users' privacy. The more that is known and shared about a particular user, the more targeted advertising becomes, and the higher the fees that can be charged to advertisers.

On the other hand, users want to feel that their privacy desires are being respected and can be driven to quit a service that goes too far. Facebook in particular has run into problems time and again where it changes policy to expose more user data; does not explain the change to users' satisfaction; and makes controls for the change difficult to find, understand, and use. Thus a user commenting on a risqué picture is suddenly shocked to find her Aunt Bessie scolding her for it. Facebook's repeated, inconsiderate missteps risk it slowly turning its users against its brand.

Considerate social software makes additional sharing a carefully explained, opt-in affair. And in the case of business norms, it can be an intellectual property rights violation, so take great care.

## Social products deal appropriately with the anti-social

*Griefers* are users who abuse social systems by interfering with transactions, adding noise to conversations, and even sabotaging work. Amazon has had to deal with an entire subculture of griefers who delight in providing sarcastic reviews to goods on sale. In large-network systems with easy and anonymous account creation, griefers have little accountability for their actions and can be a major problem. Social software provides tools for users to silence griefers from ruining transactions they own, tools to categorically exclude griefers, and tools to report griefers to community caretakers. The trick in these tools is to help caretakers distinguish the genuinely anti-social user from those who are earnest but unpopular.