Out of context: Computer systems that adapt to, and learn from, context

by H. Lieberman
T. Selker

There is a growing realization that computer systems will need to be increasingly sensitive to their context. Traditionally, hardware and software were conceptualized as input/output systems: systems that took input, explicitly given to them by a human, and acted upon that input alone to produce an explicit output. Now, this view is seen as being too restrictive. Smart computers, intelligent agent software, and digital devices of the future will have to operate on data that are not explicitly given to them, data that they observe or gather for themselves. These operations may be dependent on time, place, weather, user preferences, or the history of interaction. In other words, context. But what, exactly, is context? We look at perspectives from software agents, sensors, and embedded devices, and also contrast traditional mathematical and formal approaches. We see how each treats the problem of context and discuss the implications for design of context-sensitive hardware and software.

We are in the middle of many revolutions in computers and communication technologies: ever faster and cheaper computers, software with more and more functionality, and embedded computing in everyday devices. Yet much about the computer revolution is still unsatisfactory. Faster computers do not necessarily mean more productivity. More capable software is not necessarily easier to use. More gadgets sometimes cause more complications. What can we do to make sure that the increased capability of our artifacts actually improves peoples' lives?

Several subfields of computer science propose paths to a solution. The field of artificial intelligence tells us that making computers more intelligent will help. The field of human-computer interaction tells us that more careful user-centered design and testing of direct-manipulation interfaces will help. And indeed they will. But in order for these solutions to be realized, we believe that they will have to grapple with a problem that has previously been given short shrift in these and other fields: the problem of context.

We propose that a considerable portion of what we call intelligence in artificial intelligence or good design in human-computer interaction actually amounts to being sensitive to the context in which the artifacts are used. Doing "the right thing" entails that it be right given the user's current context. Many of the frustrations of today's software—cryptic error messages, tedious procedures, and brittle behavior—are often due to the program taking actions that may be right given the software's assumptions, but wrong for the user's actual context. The only way out is to have the software know more about, and be more sensitive to, context.

Many aspects of the physical and conceptual environment can be included in the notion of context. Time and place are some obvious elements of context. Personal information about the user is part of context: Who is the user? What does he or she like or dislike? What does he or she know or not know?

Figure 1 The traditional computer science "black box"

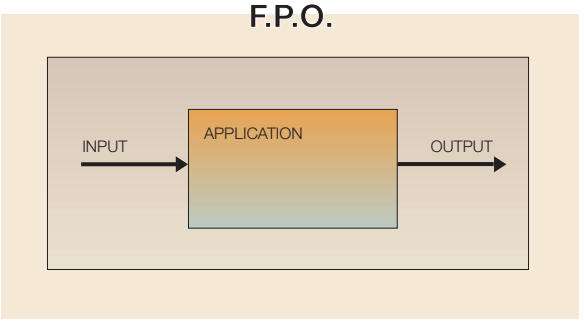
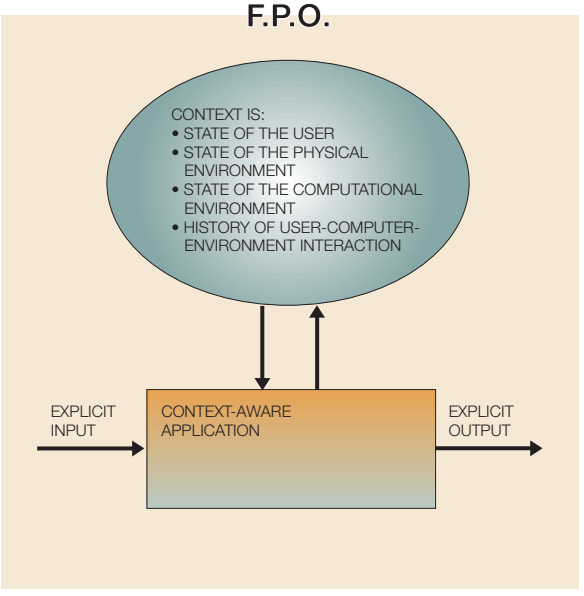


Figure 2 Context is everything *but* the explicit input and output



History is part of context. What has the user done in the past? How should that affect what happens in the future? Information about the computer system and connected networks can also be part of context. We might hope that future computer systems will be self-knowledgeable—aware of their own context.

Notice how little of today's software takes any significant account of context. Most of today's software acts exactly the same, regardless of when and where and who you are, whether you are new to it or have used it in the past, whether you are a beginner or an expert, whether you are using it alone or with friends. But what you may want the computer to do could be different under all those circumstances. No wonder our systems are brittle.

What is context? Beyond the "black box"

Why is it so hard for computer systems to take account of context? One reason is that, traditionally, the field of computer science has taken a position that is antithetical to the context problem: the search for *context-independence*.

Many of the abstractions that computer science and mathematics rely on—functions, predicates, subroutines, I/O systems, and networks—treat the systems of interest as black boxes. Something goes in one side, something comes out the other side, and the output is completely determined by the input. This is shown in Figure 1. We would like to expand that view to take account of context as an implicit input and output to the application. That is, the application can decide what to do, based not only on the explicitly presented input, but also on the context, and its result can affect not only the explicit output, but also the context. Context can be considered to be everything that affects the computation except the explicit input and output, as shown in Figure 2.

And, in fact, even this diagram is too simple. To be more accurate, we should actually close the loop, bringing the output back to the input. This acknowledges the fact that the process is actually an iterative one, and a state that is both input to and generated by the application persists over time and constitutes a feedback loop.

One consequence of this definition of context is that what you consider context depends on where you draw the boundary around the system you are considering. This affects what you will consider explicit and what you will consider implicit in the system. When talking about human-computer interfaces, the boundary seems relatively clear, because the boundary between human and computer action is sharp. Explicit input given to the system requires explicit user interface actions—typing or menu or icon selection in response to a prompt, or at the time the user expects the system's actions to occur. Anything else counts as context—history, the system's use of file and network resources, time and place if they matter, etc.

If we are talking about a internal software module, or the software interface between two modules, it

becomes less clear what counts as context, because that depends on what we consider "external" to that particular module. Indeed, one way that computer scientists sometimes deal with troublesome aspects of context is through "reification"—redrawing the boundaries so that what was formerly external to a system becomes internal. We must always be clear about where the boundaries of a system are. Anything outside is context, and it can never be made to go away completely.

The context-abstraction trade-off. The temptation to stick to the traditional black-box view comes from the desire for abstraction. Mathematical functions derive their power precisely from the fact that they ignore context, so they are assumed to work correctly in all possible contexts. Context-free grammars, for example, are simpler than context-sensitive grammars and so are preferable if they can be used to describe a language. Side effects in programming languages are changes to or dependencies on context, and they are shunned because they thwart repeatability of computation.

Thus, there is a trade-off between the desire for abstraction and the desire for context sensitivity. We believe that the pendulum has now swung too far in the direction of abstraction, and work in the near future should concentrate more on reintroducing context sensitivity where it is appropriate. Since the world is complex, we often adopt a divide-and-conquer strategy at first, assuming the divided pieces are independent of each other. But a time comes when it is necessary to move on to understanding how each piece fits in its context.

The reason to move away from the black-box model is that we would like to challenge several of the assumptions that underlie this model. First, the assumption of explicit input. In user interfaces, explicit input from the user is expensive; it slows down the interaction, interrupts the user's train of thought, and raises the possibility of mistakes. The user may be uncertain about what input to provide, and may not be able to provide it all at once. Most of us are familiar with the hassle of entering the same information many times into forms on the Web. If the system can get the information it needs from context (stored somewhere else, remembered from a past interaction), why does it ask for it again? Devices that sense the environment and use speech recognition or visual recognition may act on input that they sense that may or may not be explicitly indicated by the user. Therefore, in many user interface situations, the goal is to minimize input explicitly provided by the user.

Similarly, explicit output from a computational process is not always desirable, particularly when it places immediate demands on the user's attention.

Hiroshi Ishii[1] and others have worked on "ambient interfaces" where the output is a subtle changing of barely noticeable environmental factors such as lights and sounds, the goal being to establish a background awareness rather than force the user's attention to the system's output.

Finally, there is the implicit assumption that the input/output loop is sequential. In practice in many user interface situations, input and output may be going on simultaneously, or several separate I/O interactions may be overlapped. While traditional command-line interfaces adhered to a strict sequential conversational metaphor between the user and the machine, graphical interfaces and virtual reality interfaces could have many user and system elements active at once. Multiple agent programs, groupware, and multiprocessor machines can all lead to parallel activity that goes well beyond the sequential assumptions of the explicit I/O model.

Putting context in context. So, given the above description of the context problem, how do we make our systems more context-aware? Two parallel trends in the hardware and software worlds make this transformation increasingly urgent. On the hardware side, smaller computation and communication hardware and less expensive sensors and perceptual technologies make embedded computing in everyday devices more and more practical. This gives the devices the ability to sense the world around them and to act on that information. But how? Devices can easily be overwhelmed with sensory data, so they must determine what is worth acting on or reporting to the user. That is the challenge that we intend to meet with context-aware computing.

On the software side, we view the movement toward software agents[2,3] as an attempt to reduce the complexity of direct-manipulation screen-keyboard-and-mouse interfaces by shifting some of the burden of dealing with context from the human user to a software agent. As these agent interfaces move off the desktop, and small hardware devices take on proactive decision-making roles, we see the convergence of these two trends.

Discussion of aspects of context-aware systems as an industrial design stance can be found in a related paper,[4] which also details some additional projects in augmenting everyday household objects with context-aware computing.

In the next sections of this paper, we describe several of our projects in these areas for which we believe the context problem to be a motivating force. These projects show case studies dealing with the context problem on a practical application level and provide illustrations of the techniques and problems that arise.

We then broaden our view to very quickly survey some views that other fields have taken on the context problem, particularly traditional approaches in artificial intelligence and mathematical logic. Sociology, linguistics, and other fields have also dealt with the context problem, and although we cannot exhaustively treat these fields here, an overview of the various perspectives is helpful.

Context for user interface agents

The context problem has special relevance for the new generation of software agents that will soon be both augmenting and replacing today's interaction paradigm of direct-manipulation interfaces. We tend to conceptualize a computer system as being like a box of tools, with each tool specialized to do a particular job when it is called on by the user. Each menu operation, icon, or typed command can be thought of as being a tool. Computer systems are now organized around so-called "applications," or collections of these tools that operate on a structured object, such as a spreadsheet, drawing, or collection of e-mail messages.

Each application can be thought of as establishing a context for user action. The application determines what actions are available to the user and what objects can be operated on. Leaving one application and entering another means changing contexts—the user gets a different set of actions and a different set of objects. Each tool works in only a single context and only when that particular application is active. Any communication of data between one application and another requires a stereotypical set of actions on the part of the user (copy, switch application, paste).

One problem with this style of organization is that many things the user wishes to accomplish are not completely implementable within a single application. For example, the user may think of "arrange a trip" as a single task, but it might involve use of an e-mail editor, a Web browser, a spreadsheet, and other applications. Switching between them, transferring relevant data, worrying about things getting "out of sync," differences in command sets and capabilities between different applications, remembering what has already been done, etc., soon becomes overwhelming and makes the interface more and more complex. If we insist on maintaining the separation of applications, there is no way out of this dilemma.

How do we deal with this in the real world? We might delegate the task of arranging a trip to a human assistant, such as a secretary or a travel agent. It then becomes the job of the agent to decide what context is appropriate and what tools are necessary to operate in each context, and to determine what elements of the context are relevant at any moment. The travel agent knows that we prefer an aisle seat and how to select it using the airline's reservation system, whether we have been cleared for a waitlisted seat, how to lower the price by changing airline or departure time, etc. It is this kind of job that we will have to delegate more and more to software agents if we want to maintain simplicity of interaction in the face of the desire to apply computers to ever more complex tasks.

Agents and user intent. The primary job of the agent is to understand the intent of the user. There are only two choices: either the agent can ask the user, or the agent can infer the user's intent from context. Asking the user is fine in many situations, but making explicit queries for everything soon exhausts the user. We rely on our human agents to learn from past experience and to be able to pick up information they need from context. We expect human agents to be able to piece together partial information that comes from different sources at different times in order to solve a problem. Today's software does not learn from past interactions, always asks

explicit questions, and can deal with information explicitly presented to it only when it is ready to receive it. This will have to change if we are to make computers ever more useful.

Getting context sensitivity right is no easy task. It is particularly risky because if you get it wrong, it becomes very noticeable and annoying to the user. As in any first-generation technology, there will be occasional mistakes. As an example, the feature in Microsoft Word** that automatically capitalizes the first word of a sentence can occasionally get it wrong when you type a word following an abbreviation. The first time is not so bad. But this becomes even more frustrating as the user tries to undo the "correction" and is repeatedly "recorrected." One can take this as an argument not to do the correction at all. But perhaps the cure is more context sensitivity, rather than less. The system could notice that its suggestion was rejected by the user, and possibly also note the abbreviation so that its performance improves in the future.

Instructibility and generalization from context. All we have to start with, for humans as well as machines, is concrete experience in specific situations. For that knowledge to be of any use, it has to be *generalized*, and so generalization is key for the agent to infer the intent of the user. Generalization means remembering what the user did, and removing some of the details of the particular context so that the same or analogous experience will be applicable in different situations.

This involves an essential trade-off. A conservative approach sticks closely to the concrete experience, and so achieves increased accuracy at the expense of restricting applicability to only those situations that are very similar to the original. A liberal approach tries to do as much abstraction as possible, so that the result will be widely applicable, but at the increased risk of not being faithful to the user's original intentions.

We illustrate this relationship between generalization and context by talking about several projects that try to make software agents instructible, using the technique of "programming by example,[5] sometimes also called "programming by demonstration." This technique couples a learning agent with a conventional direct-manipulation graphical interface, such as a text or graphic editor. The agent records the actions performed by the user in the interface and produces a generalized program that can later be applied to analogous examples.

Authors in this field have noted the "data description problem," which is the problem of how to use context in deciding how much to generalize the recorded program. The system often has to make the choice of using extensional descriptions (describing the object according to its own properties) or intentional descriptions (describing the object according its role or relationships with other objects).

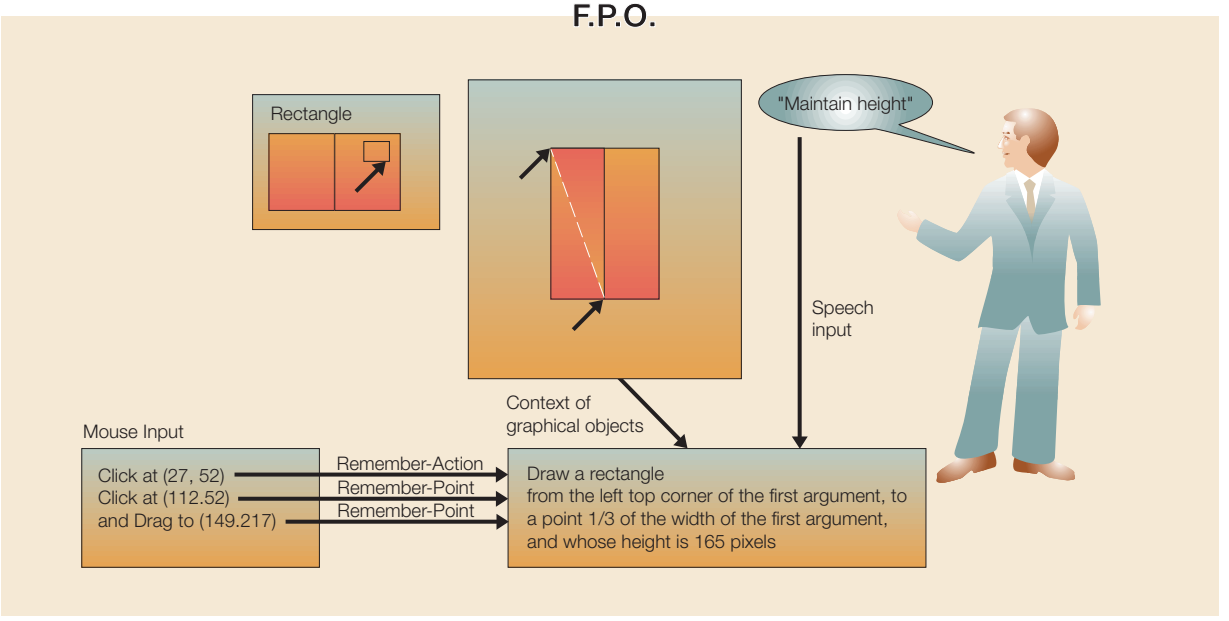
Sometimes, knowledge of the application domain provides enough context to disambiguate actions. In the programming-by-example graphical editor Mondrian,[6] the system describes objects selected by the user according to a set of graphical properties and relations. These relations are determined by extensional graphical properties (top, bottom, left, right, color) and by intentional properties of the object's role in the demonstration (an object designated as an example might be described as "the first argument to the function being defined").

We provide the user with two different ways to interactively establish a context: graphically, via attaching graphical annotations to selected parts of the picture,[6] or by speech input commands that advise the software agent "what to pay attention to"[7] (see Figure 3).

Note that neither the user's verbal instruction alone nor the graphical action alone makes sense out of context. It is only when the system interprets the action in the context of the demonstration and the context of the user's advice that the software agent has enough information to generalize the action.

User advice to a system thus forms an important aspect of context. It can be given either during a demonstration, as in Mondrian, or afterward. Future systems will necessarily have to give the user the ability to critique the system's performance after the fact. User critiques will serve to debug the system's performance, and serve as a primary mechanism for controlling the learning behavior of the system. The ability to accept critiques will also increase user satisfaction with systems, since there will be less pressure to get it right the first time. Notice that making use of critiques also involves a generalization step. When the user says "do not do *that* again," it is the responsibility of the system to figure out what "that" refers to, by deciding which aspects of the context are relevant. The ability to modify behavior based

Figure 3 Mondrian generalizes according to graphical context, user advice context, and demonstration context



on critiques is an essential difference between genuine human dialog and the rigid dialog boxes offered by today's computer interfaces.

A different kind of use of user context is illustrated by the software agent Letizia.[8] Letizia implements a kind of observational learning, in that it records and analyzes user actions to heuristically compute a profile of user interests. That user profile is then used as context in a proactive search for information of interest to the user.

Letizia tracks a user's selections in a Web browser and does a "reconnaissance" search to find interesting pages in the neighborhood of the currently viewed page. In this agent, it is analysis of the user's history that provides the context for anticipating what the user is likely to want next. Letizia shows that there is a valuable role for a software agent in helping the user to identify intersections of past context (browsing history) with current context (the currently viewed Web page and other pages a few links away from it) (see Figure 4).

Letizia illustrates a role for software agents in helping the user deal with "context overload." In situations such as browsing the Web, so much information is *potentially* relevant that the user is overwhelmed by the task of finding out what elements of the context are *actually* relevant. It is here that the software agent can step in, heuristically try to guess which of the available resources might be relevant, and put the resources most likely to be relevant at the user's fingertips.

A simple way to deal with the profusion of possible interpretations of context is for the system to compute a set of plausible interpretations and let the user choose among them. It is therefore a way for the user to give advice about context to the system.

In Grammex ("Grammars by Example"),[9] the user can teach the system to recognize text patterns by presenting examples, letting the system try to parse them, and then interacting with the system to explain the meaning of each part. Text patterns such as e-mail addresses, chemical formulas, or stock ticker symbols are often found in free text. Apple's Data Detectors[10] provides an agent that uses a parser to pick such patterns out of their embedded textual context and apply a set of predetermined actions appropriate to the type of object found. For each text fragment, Grammex heuristically produces a menu containing all the plausible interpretations of that string in its context. An example string "media.mit.edu" could mean either exactly that string, a string

of words separated by periods, or a recursive definition of a host name. Grammex illustrates how a software agent can assist a user by computing a set of plausible contexts, then asking the user to confirm which one is correct (see Figure 5).

Emacs Menus[11] is a programming environment that sits in a text editor and analyzes the surrounding text to infer a context in which a pop-up menu can suggest plausible completions. If the user is in a context where it is plausible to type a variable, the system can read the program and supply the names of all the variables that would make sense at that point. This expands the notion of context to mean not just "what is in the neighborhood right now" but, more generally, "what is typically in a neighborhood like this." That significantly improves the system's usefulness and perceived intelligence.

Models of context: User, task and system models. All computer systems have some model of context, even if it is only implicit. The computer "knows" what instructions it can process at each stage, knows what input it expects in what order, knows what error messages to give when there is a problem. Historically these have been static descriptions, represented by files or internal data structures, or encoded procedurally and used only for a single purpose. The computer's models take the form of a description of the system itself, a *system model*, the user's state, history and preferences, a *user model*, and the goals and actions intended to be performed by the user, a *task model*. To create context-aware applications, user, task and system models should best be dynamic and have the ability to explain themselves.

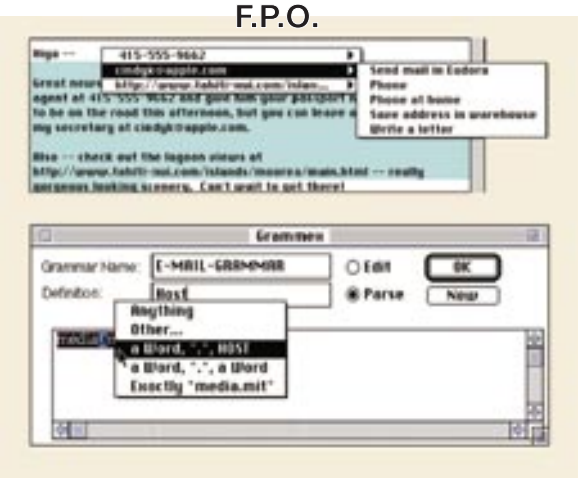
Figure 4 Letizia helps the user intersect past context (history of browsing concerning investment brokers) with current context ("Smart Money") to find "Broker Ratings".

Computer users always hold ideas of what the system is, and what they can do with it. Part of the usual system model is "if I start the right programs and type the right things, the computer will do things for me." Some users understand advanced concepts like client/server systems, or disk swap space; others do not. The system's model may change (for example, a new version of a browser might integrate e-mail). The user may or may not be aware of this integration.

Historically, computer system models are implicitly held in function calls that expect other parts of the system to be there. Better for contextual computing are systems that represent a system model explicitly, and try to detect and correct differences between the user's system model and what the system can actually do. This is analogous to "naïve physics" in physics education, where we help people understand how what they think they know, wrong as it may be, affects their ability to reason about the system. A dynamic system model could be queried about what the system could do and perhaps even change its response as it was crashing or being upgraded.

Norman[12] stressed that users frequently have models of what the system can do that are incomplete, sometimes intentionally so. They adopt strategies that are deliberately suboptimal in order to defensively protect themselves against the possibility or consequences of errors. Discrepancies between a system's assumption that the user has perfect understanding of its commands and objects, and the user's actual partial understanding, can lead to problems. This further argues for dynamic and ex-

Figure 5 Picking an e-mail address out of context and applying an action to it (top); teaching the system how to recognize an e-mail address using Grammex (bottom).



planatory models so that the user and system can come to a shared understanding of the system's capabilities.

A user's task model is always changing. Perhaps the user believes he or she needs just a simple calculation to complete some work. If the result is different than expected, the user's task model should change. The computer also has expectations (e.g., "a user would never turn me off without shutting down . . . "). A typical graphical help system uses a static task model. "Wizards" typically assume that the user will always do things in a certain way. The wizards in Microsoft Windows** take a user through a linear procedure. Any change from the shown procedure is not explained. So wizards do not help the user learn to generalize from a specific situation. Better approaches can actually use the context to teach concepts that improve user understanding and future performance.

The COgnitive Adaptive Computer Help (COACH) system (discussed later) has a taxonomy of user tasks. This taxonomy allows the system to have a dynamic task model. Sunil Vemuri[13] is building a system called "What Was I Thinking?" that records segments of speech, and, without necessarily completely understanding the speech, maps the current segment onto a similar segment that occurred in the recent past. This system expects that users have different tasks. Computer programs that anticipate changing uses are more context-aware.

The computer has a model of what it thinks the user can do: the user model. In most cases this model is that the user knows all of its commands and should enter them correctly. Explicit user models have been proposed and used for some time. The Grundy system[14] used a *stereotype*, a list of user attributes such as age, sex, and nationality, to help choose books in a library. Such a stereotype is an example of a user model that allows a computer to take user context into account.

"Do What I Mean" or DWIM[15] incorporated a system model that would change as a person wrote a program, tracking the program variables and functions. DWIM would notice when a person typed a function name that had not been defined. It would act as though it believed the person's intended task was to type a known function name and would look for a similar defined name. DWIM used a dynamic user model to search through user-defined words for possible spelling analogs that might be intended. Unfortunately, DWIM also made some bad decisions. This has been corrected in modern successors, which interactively display suggested corrections and permit easy recovery in case of wrong guesses. Early in the 1990s Charles River Analytics' Open Sesame** tracked user actions in Apple's Macintosh** operating system and offered predictive completion of operation sequences, such as opening certain windows after opening a particular application.

However, just having a user model does not ensure an improved system. In the 1970s and 80s, Sophie[16] and other systems attempted to drive an electronic teaching system from an expert user model. It was found that novice users could not be modeled simply as experts with some missing knowledge. The things that a user needs to know has more to do with the user than with the expert he or she might become.[17]

The COgnitive Adaptive Computer Help (COACH) system,[18] uses adaptive task user and system models to improve the kind of help that can be given. This system was to help users actually improve their ability to learn to program. As well as a dynamic system model, COACH tracked user experience and expertise and used it to decide what help to give. The system recorded which constructs were used, how often, and whether the user's command was accepted by the computer, adding usage examples and error

examples from other users' experiences. COACH also added user-defined constructs to the system model, so that the user can explicitly teach the computer.

Context-dependent help is help that is relevant to the commands and knowledge currently active. Relative to our notions of context, such help systems are using a system model to make help appropriate to the situation at the time of the query. Not surprisingly, such context-dependent or integrated help has been shown to improve users' ability to utilize it.[19]

COACH is a proactive, adaptive help system that explains procedures in terms of the user's own context. In contrast to help systems or wizards that use one example for one problem, COACH explains its procedures using the present context. COACH was implemented first for teaching any programming language that could be typed through an Emacs-like editor or C shell command line. A user study demonstrated that COACH's adaptive contextual help could improve LISP students' ability to learn and write LISP. COACH models each task at the novice, intermediate, professional, and expert level.

Later versions of COACH were deployed as OS/2* (Operating System/2*) Smart Guides. This used graphical, animated, and audio commentary to teach users about many of the important GUI (graphical user interface) interaction techniques. Figure 6 shows COACH demonstrating a drag interaction. We call this image a *slug trail*, which marks the important visual context surrounding when to press, move, and lift up on the mouse. Another technique developed for COACH is the *mask*, a graphical annotation creating a see-through grid that highlights important selectable items in the context. COACH adaptively creates the mask on the user interface.

Figure 6 COACH explains operations in terms of the user model ("Level 1" at bottom), system model (maskind disabled at top), and task model (icon acted upon and state of mouse at bottom)

Context for embedded computing

The toilet flushes when the user walks away. The clock tells us it is time to get up. These are all examples of context awareness with no user typing into a computer. When computers can sense the physical world, we might dispense with much of what is done with keyboards and mice. Using knowledge of what we do, what we have done, where we are, and how we feel about our actions and environment is becoming a major part of the user interface research agenda.

People say many things: It is what they do that counts. One of the obvious advantages of context-aware computing is that it does not rely on the symbolic. Symbolic communication must be interpreted through language; communication through context focuses on what we actually do and where we are.

Individuals often communicate multiple messages simultaneously that might be hard to separate. Messages often communicate things about us (age, health, sleepiness, social interest in each other, state of mind, priority of this communication, level of organization, level in the organization, social background, etc.) as well as their ostensible content.

Human speech tend to be full of errors. We communicate what we think should be interpretable, but often underspecify in the utterance. Speech is by nature unrelated to its physical subject (persons, places, things); without feedback it is hard to know how our

speech is perceived. Our logic, or the correspondence of what we are saying with the thing we are talking about, is often flawed.

The modality of verbal communication is often less dense than direct observation of physical acts. Describing what part of something should be observed or manipulated in a particular way can be quite cumbersome compared to actually doing it and having an observer watch. Some things are easier done than said.

Things in your head become things in your life. With the advent of ever smaller, faster, and cheaper computing and communications, computing devices will become embedded in everyday devices: clothing, walls, furniture, kitchen utensils, cars, and many different kinds of handheld gadgets that we will carry around with us. Efforts such as the MIT Media Laboratory's Things That Think and Wearable Computing projects, Xerox Parc's Ubiquitous Computing,[20] Motorola's Digital DNA, and others, are aimed toward this future. It is our hope that these devices will enhance our lives and not be an annoyance, but that will depend on whether or not the devices can take the action that is appropriate to the context in which they will be used.

Everyone finds cell phones a convenience until they ring inappropriately at a restaurant. Phone companies bristle at the fact that people habitually keep cell phones turned off to guard against just this sort of intrusion. Of course, phones that vibrate, rather than ring, are one simple solution that does not require context understanding and illustrates how sometimes a simple context-free design can work. But even the vibration puts you in a dilemma about whether to answer or not. Potentially, a smarter cell phone of the future could have a GPS (Global Positioning System), know that it is in a restaurant, and take a message. Or the phone could respond differently to each caller, know which callers should immediately be put through, and which could be deferred.

Context will be useful in cutting down the interface clutter that might otherwise result from having too many small devices, each with its own interface. Already, the touch-tone interface to a common office phone is getting so overloaded with features that most users have difficulty. Specializing "information appliance" devices to a particular task, as recommended by Norman[21] simplifies each device, but leaves the poor user with a proliferation of devices, each with its own set of buttons, display, power supply, user manual, and warranty card. As we have noted, context can be a powerful factor in reducing user input, in embedded computing devices as well as desktop interfaces.

Interfaces for physical devices put different constraints on user interaction than do screen interfaces. Display space is small, if any exists, and space for buttons or other interaction elements is also restricted or may not exist. Users need to keep attention focused on the real-world task, not on interacting with the device.

Transcription: Translating context into action. Perhaps the greatest potential in embedded sensory and distributed computing is the possibility of eliminating many of the *transcription* tasks that are otherwise foisted on users. Transcription occurs when the user must manually provide, as input, some data that could be collected or inferred from the environment. Transcription relies on the user to perform the translation. This simple act frequently introduces errors. Ergonomic realities mean that explicit transcription is bound to be more stressful than transferring information implicitly. Mitch Steinhas also noted the centrality of the transcription problem,[22] and promoted Krishna Nathan's work at IBM on the CrossPad** product in response to the handwritten-notes-to-typing transcription that people often perform. Transcription can also take the form of translating a user's intuitive goal into a formal language that the computer understands; the most extreme example is the translation of procedural goals into a programming language. The difficulty of learning new computer interfaces, and of programming itself, is largely traceable to the cognitive barriers imposed by this kind of transcription.

"Smart" devices that sense and remember aspects of the surrounding context are becoming interfaces to computational elements, improving human-com-

puter relationships. These devices will be able to listen to and recognize speech input, perform simple visual recognition, and perhaps even sense the emotional state of the user, via sensors being developed in Rosalind Picard's Affective Computing project.[23] Within a very short time, all desktop software, including Internet accessors, will record parts of their human interface; sensors will record workers' actions in offices, labs, and other environments. The challenge for the software will be to determine what parts of the context are relevant.

At the MIT Media Lab's Context-Aware Computing Lab, we have developed several interfaces that illustrate the power of automatically sensing context to eliminate unnecessary transcription tasks. These fall into two categories. The first is context-aware devices that augment the static environment, such as intelligent furniture with embedded computing and interaction capabilities. The second is wearable, portable, or attachable devices that augment the users themselves.

The simplest project in the area of intelligent furniture is the Talking Couch, developed at IBM Almaden's USER group. A couch positioned in a lobby often serves the purpose of inviting the user to take a break to wait for something to happen. Usually magazines are on the tabletop; a TV might be on in the corner. The digital couch does more—it orients the visitor, suggesting what he or she could do during this break. It speaks as its occupant sits down, informing the visitor about what is going on, what time it is, and what he or she might do. It announces when the scheduled conference has its next break, when the cafeteria might be open, and who the next speaker is. If the occupant is wearing a specially designed personal digital assistant (PDA), the couch goes onto point out specific user information that could be relevant: "It is always good to take a break; you have three things you said you wanted to work on when you had time." Another message might be: "In 15 minutes you have to give a talk in the auditorium." The first generic reminder message, and the more timely talk reminder message, illustrate how being reminded might be useful or irritating. The talking couch creates a user model of preferences and ambitions. Without the net-connected PDA, the couch works with the dynamic system model of its surroundings. It creates a task model: a person sitting on the couch would like to be oriented to what is going on in the area. With the PDA it adds to this a schedule-based model of the user.

Figure 7 The electric bed

Figure 8 Talking trivet

Another project instruments a bed with computing capabilities. A projection screen is mounted above the bed (Figure 7). A bed is expected to be the place for a calm, relaxing break. A projection of a sunrise on the ceiling might be nice, especially if it could act as an alarm clock, set to the time you should get up. How about going to sleep with the stars in the sky, and a constellation game to put you to sleep? If you play it too long, the game should ask if you want to get up at a later time. Projecting pages on the ceiling would allow you to read without propping yourself up on your elbows or a pillow. A multimedia bed could provide such contextually appropriate content as well as gesture recognition and postural correction and awareness.

Context can augment the mobile, as well as the static environment. A system consisting of an electronic oven mitt and trivet (the "talking trivet") uses context to transform a thermometer into a cooking safety watchdog. The talking trivet uses task models of temperatures on an oven mitt to decide how to communicate to a cook.

The talking trivet is a digital enhancement of common objects (see Figure 8). Sensing and memory in an oven mitt make it a better tool than a simple thermometer reading. The system uses a computer to take time and temperature into account in determining whether food is in need of rewarming (under 90 degrees Fahrenheit), hot and ready to eat, ready to take out (a temperature hotter than boiling water will dry food and browning starts soon after), or on fire (above 454 degrees Fahrenheit). The model is key to the value of the temperature reading. The goal concerns the importance of what the mitt is doing—it could prevent a kitchen fire. The uses of the oven mitt/trivet combination are simple; the goals of the user obviously depend on the reported temperature.

The talking trivet could well be in a better position to know when to take a pizza out of an oven than a person—it would measure the temperature when the 72-degree pizza is put in a 550-degree oven, and using its pizza model, tell the user when the pizza should be done. If the oven mitt touches the pan and finds it to be only 100 degrees after 10 minutes, it should go back to its contextual model and reflect that this item must be much more massive than a pizza. It might express alarm to the user: 550 degrees is too hot for a roast! This example underscores the value of a task model in a contextual object. The talking trivet need not be "told" anything explicit. It can act as a fire alarm, cooking coach, and egg timer, based only on what it experiences and its models of cooking and the kitchen.

The view of context from other fields

Many other fields have treated the problem of context. In what follows, we present a little of our perspective on how other fields have viewed the context problem.

Mathematical and formal approaches to AI. Several areas of mathematics, and formal approaches to artificial intelligence (AI), have tried to address context in reasoning. When formal axiomatizations of common-sense knowledge were first used as tools for reasoning in AI systems, it quickly became clear that they could not be used blindly. Simple inferences: "If Tweety is a bird, then conclude that Tweety can fly" seemed plausible until the possibility that Tweety might be a penguin or an ostrich, a stuffed bird, an injured bird, a dead bird, etc., was considered. It would be impossible to enumerate all the contingencies that would make the statement definitive.

McCarthy[24] introduced the idea of circumscription as a way to contextualize axiomatic statements. Like many of the formal approaches that try to deal with the problem, this technique gives to each logical predicate an extra argument to represent the context. The notation tries to make this extra argument implicit to avoid complicating proofs that use the technique. Then we could say, using common sense reasoning, "If X is a bird, then assume X can fly, unless something in the context explicitly prevents it." This is quite a hedge!

In artificial intelligence, researchers have identified the so-called "frame problem." In planning and robotics systems that deal with sequences of actions, each action is typically represented as a function transforming the state of the world before the action to the state after the action. The frame problem is to determine which statements that were true before the action remain true after the action, or how the action affects and is affected by its context. Solving the frame problem requires making inferences about relevance and causal chains.

In traditional mathematical logic, statements proven true remain true forever, a property called *monotonicity*. However, the addition of context changes that, since if we learn more about the context (for example, we learn that Tweety recently died), we

might change our minds. Nonmonotonic logic studies this phenomenon. A standard method for dealing with nonmonotonicity in AI systems is the so-called "truth maintenance system," which records dependencies among inferences and can retract assertions if all the assumptions on which they rest become invalid.

Even traditional modal logics can be seen as a reaction to the context problem. Modal logics introduce quantifiers for "necessary" and "possible" truths, and are typically explained in terms of possible world semantics. Something is necessary if and only if it is true in every possible world, and possible in case it is true in at least one world. Each possible world represents a context, thus modal logics enable reasoning about the dependence of statements on context.

A continuing issue in AI is also the role of background knowledge or "common sense" knowledge as context. A controversial position, probably best exemplified by Doug Lenat's CYC project,[25] maintains that intelligence in systems stems primarily from knowing a large number of simple facts, such as "water flows downhill" or "if someone shouts at you, she is probably angry with you." The intuition is that even simple queries depend on understanding a large amount of context: common-sense knowledge that remains unstated, but is shared among most people with a common language and culture. The CYC project has attempted for more than ten years to codify such knowledge, and has achieved the world's largest knowledge base, containing more than a million facts. However, the usability of such a large knowledge base for interactive applications such as Web browsing, retrieval of news stories, or user interface assistants has yet to be proven. You can get knowledge in, but it is not so easy to get it out.

The CYC approach could be labeled the "size matters" position. It could also be considered an outgrowth of the expert systems movement of the 1980s, where systems of rules for expert problem-solving behavior were created by interviewing domain experts, and the rule base matched to new situations to try to determine what to do. Expert systems were brittle; since there was no explicit representation of the context in which the expertise was situated, small changes in context would cause previously entered rules to become inapplicable. Many researchers at Stanford, including John McCarthy and Mike Genesereth, worked on axiomatic representations of common-sense knowledge and theorem-proving techniques to make these representations usable.

AI is now turning toward approaches in which large amounts of context are analyzed, both through knowledge-based methods and statistically, to detect patterns or regularities that would enable better understanding of context. Data mining techniques can be viewed in this light. Data mining is a knowledge discovery technique that analyzes large amounts of data and proposes hypothetical abstractions that are then tested against the data. The Web has also encouraged the rise of information extraction[26] techniques, where Web pages are analyzed with parsers that stop short of complete natural language understanding. The parsers approximate inference using techniques such as TFIDF (term frequency times inverse document frequency) keyword analysis, latent semantic indexing, lexical affinity (inferring semantic relations from proximity of keywords), part-of-speech tagging and partial parsers. The availability of semantic knowledge bases such as WordNet[27] also encourages partial understanding of context expressed in natural language text.

Finally it is worth noting that there is a dissenting current in AI that decries the use of any sort of representation, and therefore denies the need for context-aware computing. This position is best represented in its most extreme form by Rodney Brooks,[28] who maintained that intelligence could be achieved in a purely reactive mode, without any need for maintaining a declarative representation. Abstraction is built up only by a subsumption architecture, where sets of reactive behaviors are successively subsumed by other reactive behaviors with greater scope.

It should be clear from the previous discussion that our position on context places emphasis on shared understanding of context between humans and machines, growing as both the computer and the user observe and understand mutual interaction. We believe that even though some knowledge may be built in in advance, it is impossible to assume, as CYC and the Stanford axiomatic theoreticians do, that most or all can be codified in advance. We also reject the Brooks position that representation and context is unnecessary for intelligence. We would like to structure an interaction so that users teach the system just a little bit of context with each interaction, and the system feeds back a little bit of its understanding at each step. We believe that in this way, representations of both common-sense and personal context can be built up slowly over time.

Context in the human-computer interface field. Context plays a big role in information visualization and visual design in general. Tufte[29] and other authors have noted that the choice of visual appearance of an interface element should depend on its context, since human perception tends to pick up similarities of color, shape, or alignment of objects. Visual similarity in a design implies semantic similarity, whether it is intentional or not. A visual language of interface design needs to consider these relationships, and automatic design tools can be designed that automatically map semantic relationships into visual design choices.[30,31]

Introductory texts on user interface design stress the importance of interaction with end users. Designers are admonished to ask users what they want, testing preliminary mock-ups or low-fidelity prototypes early in the design process. User-centered and participatory design practices have been especially widespread in Scandinavia. This gives the interface designers the best understanding of the users' context in order to minimize mismatches between the designers' and the users' expectation.

Context in sociology and behavioral studies. Approaches in sociology have stressed the importance of context in observing how people behave and in understanding their cognitive abilities. Lucy Suchman and colleagues[32] have championed what they call the *situated action* approach[33] that stresses the effect of shared social context on human behavior. However, the situated critique focuses on getting system designers to adapt designs to context, and not on having the system itself dynamically adapt to context.

Another relevant field is the *activity theory* approach,[34] growing out of a Russian psychology movement. Other related fields, such as industrial systems engineering, ecological psychology, ethology, and cognitive psychology, also study context. They investigate how the contexts of behavior are critical for determining both what constitutes successful behavior and what strategies an agent must employ to generate that behavior.

Probably the most striking work in understanding how context affects human interaction with computers is Clifford Nass and Byron Reeves' Media Equation.[34] Ingeniously designed experiments dramatically demonstrate how individuals transfer human social context into interaction with machines, voluntarily or not.

Simplifying interfaces without "dumbing them down"

Using computers takes too much concentration. It takes significant time to learn and deal with computer interaction, rather than the task one is attempting. Individuals must switch contexts, from thinking about what they are interested in to thinking explicitly about what commands will have the effects they intend. The danger is that the presence of computers may distract from direct experience. It is similar to an eager relative, so engrossed in taking pictures at a beach party that we wonder if they are truly experiencing the beach.

Context-aware computing gives us a way out of this dilemma. Tools can get in the way of tasks, and context-aware computing gives us the potential for taking the tool out of the task. When computers or devices sense automatically, remember history, and adapt to changing situations, the amount of unnecessary explicit interaction can be reduced, and our systems will be more responsive as a result.

We want simpler interfaces. But if the only way we can get simpler interfaces is to reduce functionality, we "dumb down" our interfaces. Reduced functionality works well in simple situations, but can be inappropriate or even dangerous when the situation becomes more complex. Context-aware agents and context-sensitive devices can give us the sophisticated behavior we need from our artifacts without burdening the users with complex interfaces.

We have seen how software agents that record and generalize user interactions, and sensor-based devices that provide context-appropriate behavior, hold the potential for getting us off the treadmill of added features. It is now time to integrate what we have learned about context, both from the mathematical and sociological fields in which it has been traditionally studied, and from the new perspective that comes from AI and human-computer interfaces, to work toward the effortless success we always dream of.

Cited references

1. C. Wisneski, H. Ishii, A. Dahley, M. Gorbet, S. Brave, B. Ullmer, and P. Yarin, "Ambient Displays: Turning Architec-

tural Space into an Interface Between People and Digital Information," *Proceedings of First International Workshop on Cooperative Buildings (CoBuild '98)*, Darmstadt, Germany (February 25–26, 1998), pp. 22–32.
2. *Software Agents*, J. Bradshaw, Editor, AAAI Press/MIT Press, Menlo Park, CA (1997).
3. P. Maes, "Agents That Reduce Work and Information Overload," *Communications of the ACM* **37**, No. 7, 30–40 (July 1994).
4. T. Selker and W. Burleson, "Context-Aware Design and Interaction in Computer Systems," *IBM Systems Journal* **39**, Nos. 3&4, ●●●–●●● (2000, this issue).
5. *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor, MIT Press, Cambridge, MA (1993).
6. H. Lieberman, "Mondrian: A Teachable Graphical Editor," *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor, MIT Press, Cambridge, MA (1993).
7. E. Stoehr and H. Lieberman, "Hearing Aid: Adding Verbal Hints to a Learning Interface," *Proceedings of the Third ACM Conference on Multimedia*, San Francisco, CA (November 5–9, 1995).
8. H. Lieberman, "Autonomous Interface Agents," *ACM Conference on Human Factors in Computer Systems*, Atlanta, GA (March 22–27, 1997), pp. 67–74.
9. H. Lieberman, B. Nardi and D. Wright, "Training Agents to Recognize Text by Example," *ACM Conference on Autonomous Agents*, Seattle, WA (May 1–5, 1999). Also to appear in the *Journal of Autonomous Agents and Multi-Agent Systems* (2000).
10. B. Nardi, J. Miller, and D. Wright, "Collaborative, Programmable Intelligent Agents," *Communications of the ACM* **41**, No. 3, 96–104 (March 1998).
11. C. Fry, "Programming on an Already Full Brain," *Communications of the ACM* **40**, No. 4, 55–64 (April 1997).
12. D. A. Norman, "Some Observations on Mental Models," *Mental Models*, D. Gentner and A. L. Stevens, Editors, Lawrence Erlbaum Associates, Hillsdale, NJ (1983), pp. 15–34.
13. S. Vemuri, personal communication (1999). Also see http://context99.www.media.mit.edu/courses/context99/.
14. E. Rich, "Users Are Individuals: Individualizing User Models," *International Journal of Man-Machine Studies* **18**, 199–214 (1983).
15. C. Lewis and D. A. Norman, "Designing for Error," *Readings in Human-Computer Interaction: A Multi-Disciplinary Approach*, R. M. Baecker and W. A. S. Buxton, Editors, Morgan Kaufmann Publishers, Inc., Los Altos, CA (1987), pp. 621–626.
16. J. Brown, R. Burton, and A. G. Bell, "Sophie: A Step Toward a Reactive Environment," *International Journal of Man Machine Studies* **7** (1975).
17. D. H. Sleeman and J. S. Brown, "Introduction: Intelligent Tutoring Systems: An Overview," *Intelligent Tutoring Systems*, D. H. Sleeman and J. S. Brown, Editors, Academic Press, Burlington, MA (1982), pp. 1–11.
18. T. Selker, "COACH: A Teaching Agent That Learns," *Communications of the ACM* **37**, No. 7, 92–99 (July 1994).
19. N. S. Borenstein, "Help Texts vs Help Mechanisms: A New Mandate for Documentation Writers," *Proceedings of the Fourth International Conference on Systems Documentation*, Ithaca, NY, (June 18–21, 1985), pp. 78–83.
20. M. Weiser, "The Computer for the 21st Century," *Scientific American* **265**, No. 3, 94–104 (September 1991).
21. D. Norman, *The Invisible Computer*, MIT Press, Cambridge, MA (1998).
22. J. Landay and R. C. Davis, "Making Sharing Pervasive: Ubiquitous Computing for Shared Note Taking," *IBM Systems Journal* **38**, No. 4, 531–550 (1999).
23. R. Picard, *Affective Computing*, MIT Press, Cambridge, MA (1997).
24. J. McCarthy, "Circumscription—A Form of Non-Monotonic Reasoning," *Artificial Intelligence Journal* **13**, 27–39 (1980).
25. D. B. Lenat and R. V. Guha, *Building Large Knowledge Based Systems*, Addison-Wesley Publishing Co., Reading, MA (1990).
26. W. G. Lehnert, "Cognition, Computers and Car Bombs: How Yale Prepared Me for the 90's," *Beliefs, Reasoning, and Decision Making: Psycho-Logic in Honor of Bob Abelson*, R. C. Schank and E. Langer, Editors, Lawrence Erlbaum Associates, Hillsdale, NJ (1994), pp. 143–173.
27. C. Fellbaum, *WordNet: An Electronic Lexical Database*, MIT Press, Cambridge, MA (1998).
28. R. A. Brooks, "Intelligence Without Representation," *Artificial Intelligence Journal* **47**, 139–159 (1991).
29. E. Tufte, *Visual Explanation*, Graphics Press, Hopkinton, MA (1996).
30. M. Cooper, "Computers and Design," *Design Quarterly* **142**, 22–31 (1989).
31. H. Lieberman, "Intelligent Graphics: A New Paradigm," *Communications of the ACM*, August 1996, ●●●.
32. L. Suchman, *Plans and Situated Actions*. Cambridge University Press, Cambridge, UK (1987).
33. J. Barwise and J. Perry, *Situations and Attitudes*, MIT Press, Cambridge, MA (1983).
34. *Context and Consciousness: Activity Theory and Human-Computer Interaction*, B. Nardi, Editor, MIT Press, Cambridge, MA (1995).
35. B. Reeves and C. Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*, Cambridge University Press, Cambridge, MA (1996).

General references

The Art of Human-Computer Interface Design, B. Laurel, Editor, Addison-Wesley Publishing Co., NY (1989).

P. Langley, *Elements of Machine Learning*, Morgan Kauffman, San Francisco, CA (1996).

H. Lieberman and D. Maulsby, "Software That Just Keeps Getting Better," *IBM Systems Journal* **35**, Nos. 3&4, 539–556 (1996).

H. Yan and T. Selker, "A Context-Aware Office Assistant," *ACM International Conference on Intelligent User Interfaces (IUI-2000)*, New Orleans, LA (January 9–12, 2000).

Accepted for publication May 9, 2000.

Henry Lieberman *MIT Media Laboratory, 20 Ames Street, Cambridge, Massachusetts 02139-4307 (electronic mail: lieber@media.mit.edu).* Dr. Lieberman has been a research scientist at the MIT Media Laboratory since 1987. His interests are in the intersection of computer graphics, human interface, and artificial intelligence. His current projects involve media interfaces that learn from examples presented by the user. He is a member of the Software Agents Group, which works on interface agents, intelligent assistants for interactive media applications. He has also worked with the Visible Language Workshop, which is concerned with visual design issues. From 1972 to 1987 he was a researcher at the MIT Artificial Intelligence Laboratory, where he worked on parallel object-oriented programming, knowledge representation, programming environments, machine learning, and computer sys-

tems for education. He holds a doctoral-equivalent degree (Habilitation) from the University of Paris and was a visiting professor there.

Ted Selker *MIT Media Laboratory, 20 Ames Street, Cambridge, Massachusetts 02139-4307 (electronic mail: selker@media.mit.edu).*
Dr. Selker is an MIT professor focusing on context-aware computing. Before joining the MIT faculty he worked at IBM for over a decade, created the User System Ergonomic Laboratory, and was named an IBM Fellow. During that time he also served on the faculty of Stanford University. He is recognized for the design of the "TrackPoint® III" in-keyboard pointing device, for creating the "COACH" adaptive agent that improves user performance (Warp Guides in OS/2), and for the design of the ThinkPad® 755 CV notebook computer that doubles as a liquid crystal display projector. Dr. Selker obtained his B.S. degree from Brown University, his M.S. degree from the University of Massachusetts, and his Ph.D. degrees from the City University of New York in computer science and information sciences and applied mathematics. Prior to joining IBM Research in 1985, he worked at Xerox Palo Alto Research Center, Atari Research Labs, and Stanford University, and was also a Stanford consulting professor. He is Chief Scientist for Vert Corporation, is on the Board of Directors for GetGoMail.com and is on the Board of Advisors of FindTheDot.com and Xift.