HUMAN-COMPUTER INTERACTION, 2001, Volume 16, pp. 401–419

Architectures for Context

Terry Winograd
Computer Science Department
Stanford University

ABSTRACT

The development of context-aware applications will require tools that are based on clearly defined models of context and system software architecture. This essay introduces models for each of these, examines the tradeoffs among the different alternatives, and describes a blackboard-based context architecture that is being used in the construction of interactive workspaces.

1. INTRODUCTION

Dey, Abowd, and Salber (2001 [this special issue]) offer several contributions to understanding context-aware computing. They begin by pointing out three current shortcomings in the field:

> "(a) the notion of context is still ill defined, (b) there is a lack of conceptual models and methods, … and (c) no tools are available."

Terry Winograd is Professor of Computer Science at Stanford University, where he directs the program in Human–Computer Interaction and does research on Interfaces to Digital Libraries and on Interactive Workspaces.

CONTENTS

As Dey et al. demonstrate, there is no consensus in the field on what "context" should include, and as a result it is hard to compare research directions and accomplishments across different researchers and groups. Although it is unlikely that a single definition will be accepted by all, we can work to understand the differences in approaches and how those differences shape the problems that are addressed and the solutions that are proposed.

The lack of conceptual models is addressed directly in this special issue. Dey et al. propose a "widget" model, Hong and Landay (2001 [this special issue]) propose an "infrastructure" model, and this essay presents a "blackboard" model. Through these discussions, human–computer interaction (HCI) researchers will have a variety of models to choose from and a better understanding of the tradeoffs among them.

Along with each of these models, new tools are being developed. Although they are all in early states of development, we can expect that either they or their descendants will become part of the HCI developer's "toolbox" over the next few years. This essay presents a particular set of tools that are being used in our interactive workspace project, and illustrates their use.

2. THE *MEANING* OF CONTEXT

2.1. Context in Language

The notion of "context" has been adapted to computing from its original use referring to language, which is reflected in the structure of the word itself: *con* (with) *text*. In using language we produce a *text*, either written or oral, intended to be interpreted by one or more other people. That text is not an encapsulated representation of an intended meaning, but rather is a cue that allows the anticipated audience to construct an appropriate meaning. That construction, in turn, is heavily based on what goes with the text: the context.

Consider the sentence "Yes, but hit the next one harder." The words carry meanings, but at every level of linguistic analysis, the speaker's intention can only be determined through inferences based on context The other what? Why "but"? What is the "Yes" a response to? Even the meaning of individual lexical items (such as "hit") cannot be determined out of context. Are we talking here about hammering a nail, or making points in a talk?

Linguists and philosophers have devoted a great deal of thought to identifying the elements of context that determine meaning. Attempts to build computer programs that can understand natural language have foundered on the complexities of knowledge and inference that are required to handle context in normal human discourse. From this effort, a few basic points have become clear.

1. Context is not just more text. Many explorations of context have focused on the text that goes with the text to be interpreted. It is possible that the sentence preceding our example was "Did that nail go in all the way?," in which case the "next one" is known to be a nail. On the other hand, the preceding text could just as well have been "Was that OK?," in which case the object being hit might be understood from the setting without ever being mentioned.
2. Context is effective only when it is shared. Context in the linguistic sense is a feature of communication. It doesn't make sense to simply "be in a context," but rather we apply context in the process of interpretation. This puts it into the consensual space between speaker and hearer, or what Clark (1996) called "common ground."
3. Context emerges in dialog. The shared interpretation of context is built around commonality of physical setting (or its tele-electronic equivalent), but it extends beyond what can simply be seen and heard. For example, the fact that one person is trying to achieve some goal may be part of the shared context, which emerges as a result of the dialog between the two parties.

2.2. Context in Human–Computer Dialog

It is not surprising that in designing mechanisms for people to interact with computers, we have taken advantage of the natural human ability to interpret context. Although the popular mantra is that computers are "literal minded" and "you have to tell them everything," this is in fact not at all the case in many common forms of interaction.

Consider the question "What will happen when you hit the key marked 'backspace' on the keyboard?"

One possible answer is "The alphabetic character immediately to the left of the blinking marker on the screen will disappear and the marker will move to the left over where it was." But an equally valid answer could be "A large rectangular area of the displayed photograph will turn bright purple." How can the same key have such disparate meanings? The reason, of course, is context.

When you sit in front of a workstation with a graphical user interface (GUI), a large amount of contextual information is used in the execution of actions by the machine. There is a collection of "running applications," one of which has "window focus." Depending on the application, that window may have a "current selection," and other properties, such as the "current background color." If the running application is PhotoShop, if the focus window contains a rectangular selected area, if the image is in a suitable mode, if an appropriate layer is selected, and if the current background color is purple, then hitting the backspace key means "replace that rectangular area with purple."

There are direct analogies with linguistic mechanisms such as pronouns, definite referring phrases, domain-specific interpretation of word senses, and so forth. The intended meaning is based on the explicit text, together with the mutual understanding of context. This context-dependency is not a special property of interfaces or something new in computing. It is a consequence of the fact that we have created a situation of communication. Context takes different forms in command-line interfaces, GUIs, speech interfaces, and so on, but the underlying phenomenon is the same for all of them: Interpretation of intention depends on mutually available context.

One of the points made earlier is that context is not just more text. In the case of the computer, it is not just more representations in the machine. Most of the context elements described earlier can be thought of as data structures somewhere in the operating system (current application, active window) and the application code (selection, background color, etc.). But consider another possible result of hitting the backspace key. A dialog box pops up on the screen with the message "Cannot delete xyz: Access is denied." Perhaps the current selection was a file on some distant file server. Then the relevant context includes whether I have an account on that server with adequate permis-

sions to delete the file with the icon I had selected. Context includes the larger world outside of the user and the system directly being used.

2.3. Context Versus Setting

Dey et al. offer the definition that context is "any information that can be used to characterize the situation of entities (i.e., a person, place, or object) that are considered relevant to the interaction between a user and the application themselves" (p. 106). This is intended to be adequately general to cover the work that has been done on context-based interaction. However, in using open-ended phrases such as "any information" and "characterize," it becomes so broad that it covers everything from the electric power grid or the list of all files on a distant server to the compiler used in creating the application.

I prefer to use "context" in a more specific way, to characterize its role in communication. Context is an operational term: Something is context because of the way it is used in interpretation, not due to its inherent properties. The voltage on the power lines is context if there is some action by the user and/or computer whose interpretation is dependent on it, but otherwise is just part of the environment. In today's computer systems, the identity of the person sitting in front of the keyboard is not part of the context. The identity typed in at login time is part of the context and is only loosely correlated with the presence of a particular person. Features of the world become context through their use.

In explaining their definition, Dey et al. elaborate with "Context is typically the location, identity, and state of people, groups, and computational and physical objects" (p. 106). This is again broad but conveys an important perspective in its emphasis on people, places, and things. The user of a computer system is always situated in some *setting* of people, places, and things (including computers), regardless of which aspects of that setting are used as context in communication. People have an informal sense of what constitutes such a setting, and much of the work on context-aware computing draws on this informal intuition. Context-aware computing might be better described as the design of computing mechanisms that can use characterizations of some standard aspects of the user's setting as a context for interaction. Note that this includes the intuitive aspect of the user's setting (places, people, and things) and also of the computer's setting (network connections and protocols, stored information, etc.).

2.4. Physical and Virtual Context

As a motivating scenario, Dey et al. describe a conference assistant. An interesting thought experiment is to rewrite their scenario with one small change. Instead of a physical conference site, imagine a Web-based conference with

speakers on video, questions via live chat, and so forth. Ashley never leaves her desk but she can still "go" to a talk, use "directions" to find the appropriate room, and find out "where" her colleagues are, all in virtual Web space.

What is of note here is that all of the features described by Dey et al. make perfect sense in this new version, but it could be fully described without any appeal to "context" or "context awareness." It does not seem novel that when you go to a Web page with a video window showing a person giving a speech, the page also displays information about the session and speaker, displays a thumbnail of the current slide, and so forth. It seems perfectly straightforward (and in fact there are commercial products) to let you take notes associated with the pages you view and to associate them with timestamps in the video presentation. Providing this functionality on a handheld device rather than a full-sized screen offers a number of design challenges, but they are challenges of information presentation, not new problems of context.

The conference assistant provides valuable functionality, but it is a mixture of very different things. The part that calls for new thinking about context and setting is the fact that actions are triggered not by clicks on a Web browser but by Ashley's physical motions from room to room. This is the domain where context awareness deals with new problems and issues. The distinction between setting-aware programs and general integration of an assistant has consequences for the kinds of tools that will be useful for building such applications. If a virtual Web version of the conference assistant is being built, the key problems are in designing information storage schemes that can link different media, including time-based media and annotations. It will be necessary to provide a format and provide storage for personal profiles so that they can be used by various parts of the application. It will be necessary to deal with changing profiles over time. All of these are important but do not require "context widgets."

On the other hand, the design of appropriate ontologies and operational conceptualizations for context elements is a major area for new research. Current systems operate as well as they do because they have evolved over time to operate in a very particular kind of setting: an individual using a single machine. The complex but now-familiar context environment of applications, windows, selections, and so forth has been hardwired into generations of operating systems and applications. People are so familiar with it that they are thrown off by even minor variations, such as the difference in handling keyboard input focus between X-Windows and Windows.

But as Dey et al. point out, we are moving away from the person-in-front-of-a-screen-and-keyboard model. People have multiple devices ranging from wearables to multipurpose mobile phones. We are building environments in which multiple users interact with each other through augmen-

tation of a variety of displays and input devices. In this new environment the old rules of context break down.

Consider a simple common case. The user makes a selection, hits CTRL-C, makes another selection (possibly in a different window or application) and hits CTRL-V, resulting in a copy of the first selection being inserted in place of the second one. The clipboard is a well-known context mechanism (this was in fact one of the great selling points of the original Macintosh interface because it allowed cross-application information movement). Clipboard use is much like the using pronouns, such as *that* and *it* in language. The commands are effectively "Copy that" and "Paste it," without needing to specify the object further.

But what happens in a multiperson, multiscreen environment? Person A does a copy of something selected on Screen 1 and does a paste on Screen 2. Between the two events, Person B copies an object on Screen 1 as well. What does "Paste it" refer to? The most recent object copied anywhere? The most recent one on the same device? The most recent one by the same person? There is no "right" answer to this question, and it is indicative of the world of complex questions that emerge when we abandon the one-user-one-machine assumptions.

It will take a new conceptual framework to address questions of this type in a coherent and unified way. Dey et al. give examples that nicely illustrate some of the questions, and it will be a challenge to future researchers to provide more general answers to them and then to build architectures and tools based on a new theoretical framework for context.

3. MODELS FOR *CONTEXT* AWARENESS

The central technical proposal of Dey et al. is the use of "context widgets" as a programming methodology. The concept is well motivated for reasons that have a long and illustrious history in software engineering. Proponents of modular program structure and object-oriented programs have long argued the advantages of separating the functionality of a component from its implementation. Writers of higher level software can call on a component using a high-level interface that abstracts away the inner details and provides a uniform way of thinking about its function.

In looking at the problem posed by the scenarios and examples in Dey et al., we see these common programming needs directly reflected. Clearly, a programmer who is writing an application, the behavior of which will depend on a user's location, should not have to be concerned with details of how location is determined: whether there is a camera-based vision system, an active badge, a magnetic tracker, or some new kind of device not yet envisioned when the program was written. The appropriate level for the module's inter-

face is one that deals in people, spatial locations, and the mappings between them.

Given a dynamically changing number of components of varying functions, it is also clear that they cannot be thought of as modules to be compiled into some grand application. Any efficiency advantages gained by the resulting close coupling are far outweighed by the complexity and fragility of the combined system. The message from distributed computing is clear. Function should be allocated to processors in whatever way best fits the setting, and the programming metaphor should be based on multiple independent communicating components.

3.1. Three Models for Context Management

A number of different organizing models have been proposed for coordinating multiple processes and components. Dey et al. propose a *widget* model, adapted from the architecture of GUIs. Hong and Landay (2001 [this special issue]) argue for an infrastructure-centered distributed services model, based on *client–server* dialog. This essay describes a third alternative, the *blackboard* model, which has been used widely in various artificial intelligence applications (Engelmore, 1988). Each of these has advantages and disadvantages, and it is useful to examine the space of tradeoffs.

Widgets

Widgets can be thought of as an extension of device drivers to the software interface. Device drivers were invented in the early days of computing to deal with the complexities of controlling hardware peripherals. Each device had its own conventions, requiring the computer program to manage a flow of data over some kind of physical connection port. Each program that used a device needed to be able to send the appropriate signals, including handling interrupts, errors, and so forth. It was obvious that as new technologies were added, this would lead to a morass. Instead, for each type of device (at some level of specificity), a standard higher level abstraction was created (e.g., files and file positioning for storage devices) and for each different physical device (e.g., a particular kind of tape drive) the operating system incorporated a driver mapping the abstraction onto the detailed control code.

A widget, such as a scroll bar on a GUI, is a device driver at a different level. The program using it can treat it as an abstract device that provides one-dimensional position information and has some additional signals (jump a screen, jump to top, etc.). The driver (widget code) can implement the functionality for any kind of pointing or wheel device, any "look and feel," and so forth. The in-

teraction is implemented in terms of messages (to the widget) and callbacks (triggering the code when a particular condition happens in the widget).

Networked Services

In traditional widget architecture, the set of active widgets belongs to some controller program (such as a window manager). Widgets are components within the manager process, rather than being implemented as independent processes. A related but more flexible model is the client–server architecture that is widely used for connecting higher level components (e.g., a user application and its database). This metaphor has served well for much of the development of Internet-based software, where the client and server reside at different net locations and communicate using Internet protocols.

In a service-based architecture, a client needs to find the location of a service (through preconfiguration or some kind of resource discovery process) and then set up a connection with it. This connection can be short-lived (e.g., the basic HTTP protocol with a single exchange) or long-lived. Connectivity is based on finding the network location (host and port) of the service and providing software that uses the service's protocol for encoding content (e.g., SQL for database queries).

A key feature of a service-based architecture is the independence of the components. There is no "widget manager" to keep global track of services and their connections. Each component contains appropriate code to create connections, marshal outgoing and incoming messages, manage failures and error messages, and so forth. This adds complexity to each component, and in turn makes them more independent. The costs of finding and communicating with independent services is inherently higher than when the components are tightly coupled in a managed process. But by using appropriately tuned specialized protocols, the code can be efficient to the degree that the underlying network latency and bandwidth allow.

As with widgets, the basic service metaphor is procedural. Each service is handled by a process on some processor. The discovery of services in a distributed environment has been a major topic of investigation (e.g., Arnold, 2000; Gribble et al., 1998). An application that needs to use a particular kind of service can either have a direct address (as is typical in configuring today's applications) or can run a discovery process with a description of the desired service. Because the search region of the discovery process can be setting-dependent (e.g., using short-range wireless), it can introduce a certain kind of setting dependency. For example, a discovery process running on a laptop looking for a printer service may be designed to find only (or preferentially) servers within the same room or building. This does not require a separate context widget but depends on the characteristics of the network connectivity and discovery services.

Blackboards

The blackboard architecture adopts a data-centric rather than process-centric point of view. Rather than sending requests to distributed components and getting callbacks from them, a process posts messages to a common shared message board, and can subscribe to receive messages matching a specified pattern that have been posted. The nature of the pattern matching varies among different blackboard systems. Artificial intelligence systems (Engelmore, 1988; Martin, Cheyer, & Moran, 1999) often apply sophisticated inference procedures to logical representations. Tuple-based blackboards, such as the early Linda language (Gelernter, 1985) and IBM's® T Spaces (Wyckoff, McLaughry, Lehman, & Ford, 1998), use simple field-by-field comparison of tuples.

In a blackboard architecture, all communications go through a centralized server. Routing to different components is effectively accomplished by the matching of message content to a subscriber's pattern. Anything that can be done with direct communication paths can be simulated in a straightforward way by including an identifier for the path (or its endpoints) as a field in a message and using matching to get messages to the desired components.

3.2. Tradeoff Criteria

In choosing an architecture, a system designer needs to consider tradeoffs along a number of different dimensions. For systems of the kind discussed in the anchor paper, these include the following:

Efficiency. All computer technology is subject to efficiency metrics in space and time. For interactive applications, the key limitations are time-efficiency, both bandwidth and latency. Some architectures make it easy to create fast paths that have been tuned for throughput efficiency, whereas others impose layers of communication structure that limit the tuning that can be done. Given today's networking and processor speeds, efficiency is not the bottleneck in many cases. For example, an application that uses information about who is in what physical space needs only a few bytes of data, and can tolerate lags measured in seconds. Dey et al.'s proposed architecture has been developed for examples of this type, rather than examples requiring highly efficient transfer of multimedia data, such as those explored by McCanne et al. (1997).

Configurability. A more difficult criterion to measure, and one that is not buoyed along by Moore's law, is the difficulty of configuring systems that include multiple processes and devices. This is often the Achilles heel

of complex system designs. Once configured, the components work effectively, but the job of adding or modifying components is complex and prone to breakdown. In many cases, changes to the configuration require a complete rebooting of the system, and cannot be done "on the fly." As computing has moved in the past few years toward a network-centric environment with dynamic addition and removal of processes, configurabiliity has become an increasing concern (see comments in Satyanarayanan, 1999, on "no-futz computing").

Robustness. A correlate of the difficulty of configuring systems is the difficulty of coping with breakdowns of their components. Traditional programming methods provide error handling mechanisms, but in general these cope only with "expected errors" and are not graceful in the face of overall component failure and disconnection. Simple error mechanisms suffice for systems in which a single process manages the set of controllers (as with standard workstation operating systems) but do not scale to systems of independent distributed components on a network. A robust system must continue to function in the face of components that malfunction, jam, send inappropriate output, disappear, and are restarted. There is no magic bullet, but the choice of architecture can aid or hinder this goal.

Simplicity. Finally, the key bottleneck is the human mind. A system that requires complex understanding by system builders to make use of its facilities will be used only by those who have the dedication and motivation to master it. The World Wide Web is an obvious object lesson. The HTML and HTTP protocols are much less powerful than many of the formatting, hypertext, and communication protocols that preceded them. But their simplicity made possible a different kind of programmer and a different arena of use. Simplicity was the key to the success of the Web.

3.3. Contrasts Between the Models

We can contrast the three models of Section 3.1 in terms of these tradeoff criteria. Rather than trying to fill out the whole matrix, we just make a few observations about the tradeoffs most relevant to the architecture of a context-aware system.

The widget model grew from a tradition of tight coupling and single-manager control. An interface with widgets is compiled together, and is an interface to one operating system. Mechanisms such as callbacks take advantage of this tight coupling for efficiency, but require complex configuration and are not robust to component failures (imagine an interface in which "the scroll bar has gone down").

The services model evolved in the Internet environment, with relatively large, independent process components each having complex functionalities. It therefore puts much less emphasis on efficiency and tight control, and correspondingly more on configurability (service discovery) and robustness.

The blackboard metaphor developed in the context of artificial intelligence systems, where each component (or "agent") had partial information and new sources could be easily added. It is the most loosely coupled and therefore pays a price in communication efficiency. Every communication requires two hops and uses a general message structure that is not optimized for any particular kind of data or interaction protocol. The benefits are in ease of configurability and robustness, and in the simplicity provided by a uniform communications path.

4. THE INTERACTIVE WORKSPACE ARCHITECTURE

In our research on Interactive Workspaces (Fox, Johanson, Hanrahan, & Winograd, 2000), we are exploring the integration of multiple devices for multiple users in a shared physical space, called the *iRoom*. We have implemented a communication and application programming architecture that supports context-aware (or "setting-aware") computing.

The overall architectural metaphor is a blackboard with two levels of data. The *Event Heap* (Fox et al., 2000), which uses T Spaces (Wyckoff et al., 1998), provides fast distribution of simple event tuples. Any process (e.g., one handling input from a switch, keyboard, motion sensor, etc.) can post tuples, which include fields for their source and timestamp, along with explicit data associated with the event type. Any process can subscribe to a pattern of field values and receive callbacks when an appropriate tuple is posted. The receiving process can remove the tuple, or can leave it for others to receive as well. We have implemented interfaces for posting and receiving events in Java™, C++, HTTP (through a proxy), and a scripting interface Each tuple has a time-to-live, so after a specified period it will automatically be deleted. The Event Heap is generally used for short-lived items, with time-to-live extending at most through an interaction session.

The second-level blackboard is the *Context Memory*, an XML-structured database that allows any process to store and retrieve XML-encoded data. This is used for data that will be relevant across applications and sessions, such as physical objects and their locations in the space, identities and properties of people, collections of files, and so forth. Queries are sent to the Context Memory as XML ASCII strings, through an HTTP interface, or by posting an event to the Event Heap with the query string as one of the fields. In addition to responses that return data in XML (using the same paths), the Memory has a

template mechanism that allows application writers to create arbitrary HTML displays for users, and to let them post new data through forms.

In choosing to use a blackboard architecture, top priority was given to the metrics of robustness and configurability. Although efficient communication is important, it is not the bottleneck in designing systems that deal with context in a general way. Given the ever-increasing speed of processors and networks, an architecture that avoids the complexity of configuring point-to-point communication paths can serve for all but a few specialized uses that require tight action–perception coupling (Winograd, 2001).

The blackboard also offers simplicity of implementation. Because there is only one standard communication link for each component, which is always the same (to the blackboard), there is no complex protocol for finding ports or resources, establishing connections, and the like. Software can achieve the logical effects of connection-based communication when needed, by using messages in a systematic way.

Robustness is effected in a more complex way. At a first level, we depend on a component that can be a single point of failure for the whole environment. This requires that it be built with the same degree of reliability as other central failure points such as the operating system and network infrastructure. Given that it is a stable component (not modified for each new device or application), this degree of reliability has turned out to be a reasonable task in our own research, and is in keeping with the general need for stable infrastructure in any system.

The benefit in return is that the loose coupling makes it easier for components to deal with the failure of other individual components (which are not usually as stable and reliable as the infrastructure). When a component fails, no communications links break other than its own link to the blackboard. Components that depend on information from the failed component can detect its absence through timeouts, if desired, and can call for a restart of a new component. We make heavy use of an announce–listen style, in which components that provide services (such as sensor data) periodically post events. A component that uses the data can listen to these events, and if one does not appear within the expected time it can initiate restart operations. Our experience has been that when a clean restart process is provided for a component, this strategy can deal well with unexpected failures of all kinds.

In addition, the centralized nature of the blackboard provides significant opportunities for system integration. A key example is the maintenance of history. If a component is providing information about the setting (e.g., the presence of people in places), it is often useful to retrieve that information for past times. If a number of components are providing such data (e.g., one widget in each building), their data needs to be jointly queried to answer a question such as "Where was Joe at 10 a.m.?" Adding state to each individual widget is not a

practical solution in many cases because it would require that the seeker of the information open connections to all the widgets (or a special aggregator for this purpose), use protocols that they support to report history, expect them all to be operative, and so forth.

The blackboard architecture is built around a database that coordinates information across the components. If a message is posted each time a person moves to a new location, a query over the set of posted messages can provide a history without the complexity and overhead of making connections and without needing to know how the data were provided. Both the Event Heap and the Context Memory offer a way to query any data that have not yet passed the time-to-live, which can be set arbitrarily by the posting process.

Similarly, it is easy to construct "observers" that subscribe to messages that are also being received by other intended components, and that can record, analyze, and monitor activity. This observer activity can be used for basic functionality (e.g., noting that a component has failed to post a message by an expected time and restarting it), or for higher level debugging, statistics gathering, logging, and problem detection. These would be extremely difficult to configure and manage if they had to tap into individual components and the multiple communication paths among them.

5. EXAMPLES

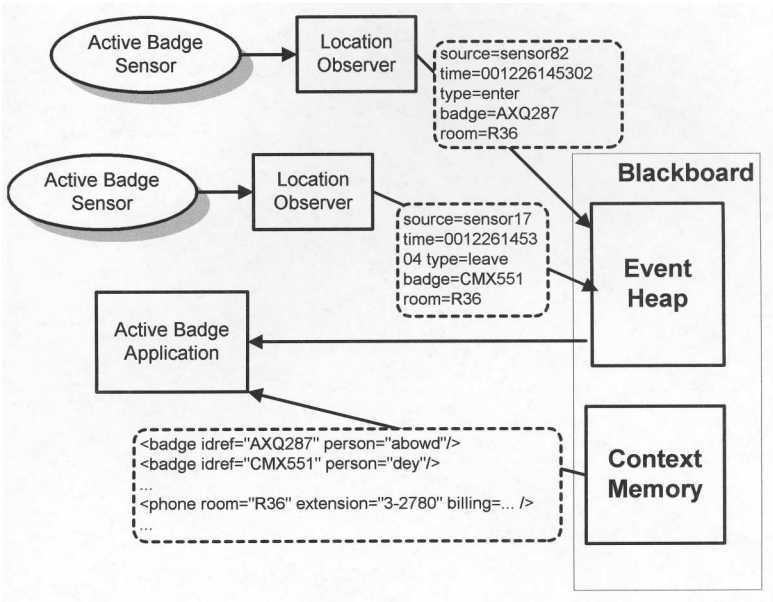
The technical examples provided by Dey at al. can be revisited from the point of view of the earlier architectural distinctions. I contrast their proposed implementations proposed with those that would be possible with the Interactive Workspace architecture.

5.1. Active Badge Call-Forwarding

Using the interactive workspace architecture, the setup for the active badge application shown in Figure 2 in Dey et al. would be replaced by the structure shown in Figure 1.

In this architecture, a variety of components introduced by Dey et al. (Interpreter, Discoverer, Aggregator, etc.) have been replaced by the shared Event Heap and Context Memory. Events are generated when a badge enters or leaves a space. These are posted to the Event Heap by a process associated with each sensor and are subscribed to by the active badge application. This application maintains information about who is where, based on the events plus heuristics about how to handle conflicting or missing data (because the sensors cannot be assumed to be always up and working properly). The active badge application does not need to deal with a collection of widgets or aggregators. Information about the assignment of badges to people and of

Figure 1. **Interactive Workspace architecture for the active badge call-forwarding application. Dotted areas show messages that are passed to and from the blackboard.**



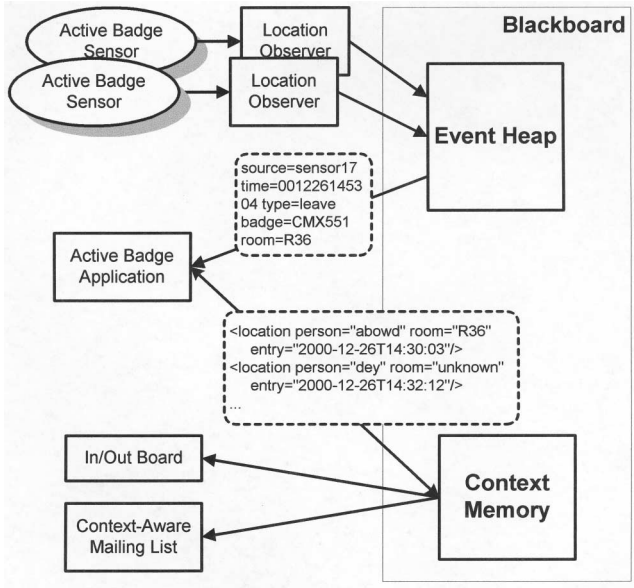
phone numbers to rooms is maintained in the Context Memory database and can be updated through standard database actions or HTML templates. The resulting system is greatly simplified, with no need to set up multiple connections, start the various widget processes (beyond the one for reading each sensor), and so forth.

5.2. In/Out Board and Context-Aware Mailing List

The example shown in Dey et al.'s Figure 7 involves two new functionalities. In adding these to our design of Figure 1, it is useful to split out the functionality of keeping track of who is where from the details of phone routing, as shown in Figure 2.

The active badge component is responsible for receiving events posted by the sensors and posting the results as a database of who is where, when, using the shared Context Memory. Any other application can use this information by querying those data. An announce–listen process can restart the active badge application if it disappears, because the relevant state is being recorded in the Context Memory. The Context Memory provides a mechanism for HTML templates that can be used with database queries, so that the produc-

Figure 2. **Interactive Workspace architecture for multiple location-aware applications. The active badge application stores a history of locations in the Context Memory, which can be queried by the other applications.**



tion of the HTML in/out board displays as shown in Dey et al.'s Figure 6 would not even require application code, but would be a template.

The point demonstrated in this example can be applied to Dey et al.'s other examples. For example, Figure 17 enumerates 26 software components that need to be configured and connected for the conference assistant. Most of these relate to a particular kind of object in the environment (names, rooms, presenters, slides, etc.). It would be more direct and simpler to think of these items not as software components but as data objects, created and manipulated by a small number of software components that share data through a common database. Many of the complexities in the examples come from the underlying architectural metaphor that approaches each item of interest in the setting as a widget.

6. RESEARCH DIRECTIONS

Dey et al. have done a service to the HCI community by drawing our attention to the question of integrating setting and context into system design at a fundamental level. Their efforts and examples have pointed out key areas of research that need to be actively pursued.

6.1. Ontologies for Context in a Distributed Environment

A truly context-aware interaction will depend on providing the application writer with a representation of the aspects of context that matter to program execution. Some of these are fairly straightforward (as with people location), whereas others are more subtle (e.g., what is the "active selection" in a multidevice, multiperson space). The research goal is to find the right level of description, which abstracts away from implementation details, but is still specific enough to serve the purpose of inferring appropriate intent from context-assuming interactions.

The hard part of this design will be the conceptual structure, not the encoding. Once we understand what needs to be encoded, it is relatively straightforward to put it into data structures, data bases, and so forth. The hard part will be coming up with conceptual structures that are broad enough to handle all of the different kinds of context, sophisticated enough to make the needed distinctions, and simple enough to provide a practical base for programming.

6.2. Robust, Simple, Distributed Architectures

The toolkit proposed by Dey et al. allows application developers to make use of distributed sources of dynamic (time-changing) data, including those that provide information about a user's physical setting. Theirs is one of many competing approaches to this problem, and a good deal of research and experimentation will be required before broadly usable designs and standards appear.

In addition to research directed to incremental improvement, we need to search for ways of cutting the Gordian Knot, as the Web protocols did with respect to their predecessors. There is no panacea: Each solution will be suited to some environments and uses but not to others. The blackboard architecture we are developing in our own research will provide robustness and simplicity for many systems, but needs to be extended for latency-critical communication, generalized to scale up to network-level environments, and augmented to provide multiple linked blackboards and protocols for managing flow among them.

6.3. User Experience

The goal of HCI design is creating an appropriate user experience. This essay, along with Dey et al., addresses the user experience only by giving some simple scenarios. Many of the other commentaries in the this special issue deal with issues that will be critical to designing and using setting-aware applications. They range from privacy and personal control to the design of "invisible interactions" in which users' intentions about what they want computers to do

are inferred from ordinary activities such as speech, gesture, and changing physical location. In an important sense, the discussion in this essay is just preparation. It deals with design of an infrastructure that will allow us to construct and test new ways of interacting. The research that begins at that point will allow us to better understand how to make setting-aware applications not just computationally feasible, but useful and appropriate.

NOTES

Acknowledgments. The implementation of the interactive workspace has been the effort of many people at Stanford. Armando Fox and Pat Hanrahan have been co-principal investigators and they along with many staff and students have contributed to the ideas. Brad Johanson was the principal developer of the Event Heap.

Support. This research has been supported in part by grants from IBM and Philips.

Author's Present Address. Terry Winograd, Department of Computer Science, Stanford University, Stanford CA 94305–9035. E-mail: winograd@cs.stanford.edu

HCI Editorial Record. First manuscript received December 22, 2000. Accepted by Tom Moran and Paul Dourish. Final manuscript received March 2, 2001. — *Editor*

REFERENCES

Arnold, K. (Ed.). (2000). *The Jini™ specifications* (2nd ed.). Reading MA: Addison-Wesley.

Clark, H. H. (1996). *Using language.* Cambridge, England: Cambridge University Press.

Dey, A. K., Abowd, G. D., & Salber, D. (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction, 16,* 97–166. [this special issue]

Engelmore, R., & Morgan, T. (Eds.). (1988). *Blackboard systems.* Reading MA: Addison- Wesley.

Fox, A., Johanson, B., Hanrahan, P., & Winograd, T. (2000). Integrating information appliances into an interactive workspace. *IEEE Computer Graphics & Applications, 20*(3), 54–65.

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems, 7*(1), 80–112.

Gribble, S. D. et al. (1999). The MultiSpace: An evolutionary platform for infrastructural services. *Proceedings of the 1999 Usenix Annual Technical Conference,* 157–170. Berkeley, CA: Usenix.

Hong, J. I., & Landay, J. A. (2001). An infrastructure approach to context-aware computing. *Human-Computer Interaction, 16,* 287–303. [this special issue]

Martin, D. L, Cheyer, A. J., & Moran, D. B. (1999). The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence, 13*(1), 91–128.

McCanne, S., Brewer, E., Katz, R., Rowe, L., Amir, E., Chawathe, Y., Coopersmith, A., Mayer-Patel, K., Raman, S., Schuett, A., Simpson, D., Swan, A., Tung, T.-L., & Wu, D. (1997). Toward a common infrastructure for multimedia-networking middleware. *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'97*), 39–49.

Satyanarayanan, M. (Ed.). (1999). Digest of proceedings. *Seventh IEEE Workshop on Hot Topics in Operating Systems*. Rio Rico, AZ: IEEE. Available: http://www.cs.rice.edu/Conferences/HotO/digest/node16.html

Winograd, T. (2001). Interaction spaces for 21st century computing. In J. Carroll (Ed.), *Human–computer interaction in the new millennium*. Reading, MA: Addison-Wesley.

Wyckoff, P., McLaughry, S. W., Lehman, T. J., & Ford, D. A. (1998). T spaces. *IBM Systems Journal, 37,* 454–474.