IEEE pervasive COMPUTING
MOBILE AND UBIQUITOUS SYSTEMS
www.computer.org/pervasive

An Architecture for Interactive Context-Aware Applications

Kasim Rehman, Frank Stajano, and George Coulouris

Vol. 6, No. 1
January–March 2007

An Architecture for Interactive Context-Aware Applications

The interactive behavior of context-aware applications, unlike that of desktop applications, depends on the physical and logical context in which the interaction occurs. A new architecture derived from the Model-View-Controller paradigm models such context in the frontend, helping users better understand application behavior.

Kasim Rehman
Cambridge Systems Associates

Frank Stajano
and George Coulouris
University of Cambridge Computer Laboratory

Many current context-aware applications suffer from three usability problems: They make inferences on the user's behalf without communicating the assumptions on which those inferences are based; they fail to provide feedback that would let users evaluate why something did or didn't work; and they don't give users control.

Although researchers are aware of such deficiencies,[1,2] we still have difficulty implementing known design principles. In our view, there are two main reasons for this. First, there has been some reluctance within the community to employ visualization for context-aware applications due to technical difficulties, [1] despite visualization's important role in Mark Weiser's original conception of ubicomp as "calm technology."[3] Second, we're missing an interaction model for context-aware application design, such as Donald Norman's Seven Stages model[4] for traditional HCI.

The main difference between traditional HCI design and context-aware design is how we deal with context. Hence, any interaction model for the latter must address the question of what context is. In our work, we pick up on recent ubicomp community trends, drawing from sociology and focusing on interaction's communicative aspects (see the Related Work sidebar for comparisons to other significant approaches).[2]

So, starting from the premise that interaction is communication, we propose a new interaction model for context-aware applications. We then derive an architectural framework that developers can use to implement our interaction model. The main benefit of our architecture is that, by modeling context in the user interface, developers can represent the application's inferences visually for users.

Research context

Our research into indoor location-aware applications motivated us to create a UI for context-aware applications. In our lab, location-aware applications support researchers in their daily interaction with computing, communication, and I/O facilities by adapting to changes in user and object locations.

However, the user experience has remained suboptimal because of the three usability problems we noted earlier. Our approach introduces visual interaction using *augmented reality* (AR). With AR, we can show users—who wear head-mounted displays—visualizations anywhere in space.[5] Visualization of context-aware applica-

1536-1268/07/$25.00 © 2007 IEEE ■ *Published by the IEEE Computer Society*

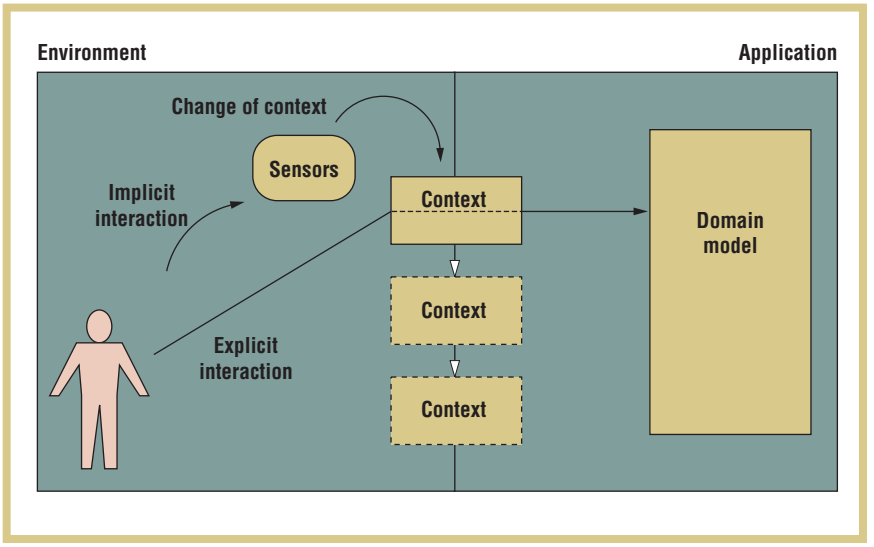


Figure 1. Our interaction model. The user's explicit interaction is always interpreted against the current context. Implicit interaction can lead to changes of context.

tions is a challenge the research community is only beginning to explore. Our prototype uses an AR interface (based on a bulky headset that introduces its own usability problems), but our architecture is independent of the visualization technology. As a result, it would work equally well with projector-based visualization, PDAs, or displays embedded in the environment.

Our location-aware applications use the SPIRIT (Spatially Indexed Resource Identification and Tracking) backend.[6] SPIRIT maintains a world model based on sensors' real-world observations. For location tracking, we use an indoor ultrasound location system that can track Active Bats (small tags). Each lab member is equipped with an Active Bat that can also act as an interaction device if its two small buttons are used. SPIRIT evaluates spatial relationships between objects and people in the world model. Applications can subscribe to be informed through events when certain relationships are fulfilled.

Our interaction model

Figure 1 shows our interaction model, which includes the physical environment (left) and the application (right). Users interact with the application while performing their daily tasks. The application is sensitive to the user's context and interprets all interaction against the current context. When a user changes context, the application reacts by changing its frame of reference for interpreting the user's actions. Such context changes can occur through *implicit interaction*, which is interaction not directly targeted at the application. For example, in a context-aware communication application, explicit interaction includes tasks such as setting up the connection, sending media, and so on, while implicit interaction consists of moving to another room.

This interpretation of what context is makes sense if you compare it with what context is in everyday life. In a conversation, for example, context isn't a piece of information about a person or object, it's a frame of reference that people use to interpret content. Similarly, in our case, explicit interaction is the content, and the context is maintained through implicit interaction.

So, what of "object context"? It appears we have only modeled user context. In our interaction model real world objects don't have "context," even though they have physical properties. For example, does a pair of computer loudspeakers sitting on a desk have context? In our view, you can't talk about context without an interpreter. So, the speakers don't have context on their own, but they can form part of the context in interactions between the user and, say, a follow-me music application. If, with such an application enabled, the user moved into the speakers' vicinity, both the user and application would see the speakers as contextually relevant and assume this understanding of each other.

In other words, we're working with a phenomenological view of context.[7] Our approach might seem alien, given the ubicomp community's prevailing positivist/engineering viewpoint, but as Paul Dourish has successfully argued, the positivist view misinterprets the way people use context in everyday life.[7] Essentially, the positivist view is concerned with representing context, whereas the phenomenological view emphasizes that people create and maintain context during interaction. And so, we believe that humans perceive context as a property of interaction, rather than of objects or people.

This distinction is important because communication between humans and context-aware applications requires a clear definition of context to be made more intelligible. In particular, our aim is to model context in the interface.

Architecture

Our architecture is based on the Model-View-Controller paradigm.[8] Figure 2 shows the MVC derivative we used to build our interactive context-aware applications. We represent the application domain using a set of models. Some of these models represent physical objects, while others represent abstract data structures that the application uses. Later, we present an example application with models for both physical objects (Active Bats) and abstract data structures, such as a list. We assume that models of physical objects can access the backend to update their states when properties of the corresponding physical objects change. In our implementation, the views are visible AR avatars of the entity mapped to the model.[5] MVC lets you represent a model in multiple ways by attaching multiple

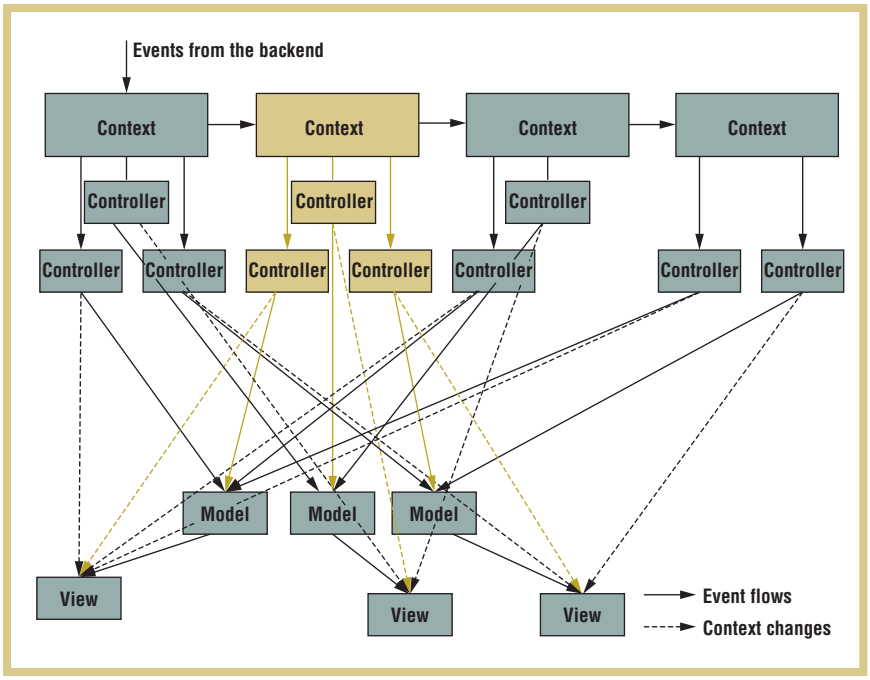


Figure 2. The context-aware application architecture. Implicit interaction events are passed on from one context component to another until a context activates itself upon receiving the event. Only the active context component (shown in gold) passes explicit interaction events to controllers.

views.[8] Although we haven't implemented this into our prototype, this architectural aspect might be exploited to support multiple alternative visualization technologies—such as AR goggles, projectors, and handheld position-sensitive PDAs serving as "lenses"—and let users quickly switch to the most convenient representation.

The top of figure 2 shows a chain of context components; the application will use one of these components for each user context it wants to react to. For location-aware computing, the user's location determines the user's context, but the context evaluation function can account for other environmental factors as well.

Implicit and explicit event handling

When context components register for events from the backend, they receive notifications about implicit and explicit interaction events (a designer-specified distinction). The backend then delivers streams of these events.

If a component receives an *implicit interaction event*, it calls its evaluation function, which then checks the current environmental conditions against the conditions that characterize the context it represents. Notably, this evaluation function can account for the environment's physical properties—that is, the properties traditionally called "context." If the function returns false, the context represented by this component is not the current one. The component thus forwards the event to the chain's next context component. This continues until one context evaluates to true. The corresponding context component then becomes active, activates its controllers, and initiates an action to deactivate all other context components. This is because, at any given time, we want the application to interpret our explicit interaction against exactly one frame of reference (context). The dotted lines in figure 2 show the references through which the activation and deactivation functions effect changes in the views.

When an *explicit interaction event* enters the chain, components forward the event through the chain until it reaches the active context component. The active context component then forwards the event down a level to its controllers. So, only controllers attached to the active context receive and interpret explicit interaction events, because all deactivated context components ignore and forward them. Explicit interaction events change model components. Such changes are instantly reflected in the view, which updates whenever the model changes.

We write one controller per model for each context modeled in the application. These controllers encapsulate how the model reacts to interaction in each context. We then attach the controllers to the corresponding context components. The key is that each controller only "wakes up" when its context becomes active. Because we use many controllers, we can separate context handling from interaction handling—the controllers contain no context evaluation code. A context can be activated again if it receives an implicit interaction event that makes its **evaluate** function evaluate to true. (Even deactivated context components monitor implicit interaction events; the components simply pass them on if they evaluate to false.)

The result is context-aware interaction modeled in architecture. Interaction is always interpreted according to its context.

What this architecture seeks to enforce

Our architecture attempts to make the communication between the user and application more intelligible in several ways.

The designer's model is communicated. When modeling the world, application designers inevitably use approximations and assumptions such as, "if X is here, he wants to do Y." In previous work, we explored how not communicating such assumptions affected usability in a well-known location-aware application.[5] When we presented a visual image of the actual model that the application de-

Related Work

Paul Dourish believes that applications should always understand context as the work practices of humans.[1] The difference between our interaction model and Dourish's model is that we've left the sociological domain and tried to apply his context interpretation to human-computer communication.

Victoria Bellotti and her colleagues proposed the idea of developing an interaction model that highlights interaction's communicative aspects.[2] In designing our model, we use their comparison of human-computer interaction with human-human interaction.

More recently, Albrecht Schmidt[3] has advocated introducing design principles to application building through "implicit human-computer interaction" (iHCI). Schmidt contrasts this with traditional HCI, wherein all interaction is explicit. Although we use this distinction of implicit and explicit interaction, our interaction model follows the conversational metaphor more closely.

Microsoft's EasyLiving project[4] also aims to build interactive context-aware applications. Although its overall aims are similar to ours, we're more concerned with understanding what context means for humans and trying to implement this in systems.

Dourish has also elaborated on the importance of communicating the designer's model.[5] He regards system design as a communication act between the designer and user that's intelligible only if there is a set of mutually agreed upon facts. For designers, "making a system usable" includes communicating "relevant aspects of the designer's model" to the user.[5]

Finally, Roy Want and his colleagues were instrumental in forwarding the context-aware computing concept.[6,7]

REFERENCES

1. P. Dourish, "What We Talk About When We Talk About Context," *Personal and Ubiquitous Computing*, vol. 8, no. 1, 2004, pp. 19–30.
2. V. Bellotti et al., "Making Sense of Sensing Systems: Five Questions for Designers and Researchers," *Proc. Computer-Human Interaction* (CHI), ACM Press, 2002, pp. 415–422.
3. A. Schmidt, "Interactive Context-Aware Systems—Interacting with Ambient Intelligence," *Ambient Intelligence*, G. Riva et al., eds., IOS Press, 2005, pp. 159–178.
4. B. Brumitt, "EasyLiving: Technologies for Intelligent Environments," *Proc. 2nd Int'l Symp. Handheld and Ubiquitous Computing* (HUC 00), LNCS 1927, Springer, 2000, pp. 12–29.
5. P. Dourish, *Where the Action Is*, MIT Press, 2001.
6. R. Want et al., "An Overview of the Parctab Ubiquitous Computing Experiment," *IEEE Personal Communications*, vol 2, no.6, 1995, pp. 28–43.
7. B. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," *Proc. 1st Ann. Workshop Mobile Computing Systems and Applications* (WMCSA), IEEE CS Press, 1994, pp. 85–90.

signer used to build the application, the users understood the application's capabilities and limitations to a degree they themselves had often not expected.

So, how can an MVC architecture help communicate the designer's model? First, it helps designers elicit the model they're using for the application by forcing them to specify the application using model objects. Second, it helps them communicate it by requiring that view objects remain faithful representations of model objects at all times. As we show later in our example, the application components effectively and elegantly communicate their operation to the user as the application reacts to the user's movement and interaction.

Context is not a piece of information, but rather a frame of reference. In our architecture, we use context to interpret interactions. Each time the context changes, the framework switches the set of active controllers, whose function is to interpret user interaction. In other words, our architecture does not implement "if *x*, do *y*," but rather, "as long as *x*, interpret all interactions taking place as relating to *x*." This is closer to our understanding of context in real life.

The current context—and entry and departure from it—is always visible. Victoria Bellotti and Keith Edwards first made this demand.[9] In our framework, we designed the controllers' activation and deactivation functions to change the view when the context changes.

The system state is always visible and instant feedback is always provided. We represent all application entities as models. Because each model has a view, the system state is always visible. More importantly, the system instantly updates the view as the system state changes.

An example of visual interaction design

Our design process consists of two parts: creating an application domain model and designing the visual interaction. Here, we concentrate on the latter because creating a domain model is part of standard object-oriented design.

Our example application is a location-aware desktop teleporting application. Using it, people can spontaneously approach any computer in our lab, push a single button on their Bats, and thereby "teleport" their personal desktops to the computer. Once the user

		A	B	C	D	E	F
		Inside teleport region			Outside teleport region		
	Controller / Function	*Bat controller*	*Desktops controller*	*Teleport service controller*	*Bat controller*	*Desktops controller*	*Teleport service controller*
1	`topBatButton Pressed()`	–	If teleporting is active, connect next desktop	–	–	–	–
2	`bottomBatButton Pressed()`	–	–	Toggle teleporting state	–	–	Toggle teleporting state
3	`batMoved(x, y, z)`	Update Active Bat position	–	–	Update Active Bat position	–	–
4	`activate()`	–	If active and connected, maximize view; If active and not connected, minimize view; If teporting is not active, deactivate view;	–	–	–	–
5	`deactivate()`	–	Deactivate view	–	–	–	–

Figure 3. Controller design. The columns (A–F) represent controllers and the rows (1–5) represent functions in those controllers that the framework invokes in response to events.

presses the button, the location-aware application running in the background infers which computer the person is going to use and connects it to a server holding the user's desktop. If users have several desktops, all are potential teleport candidates. Finally, users can override this facility by turning the teleporting service on or off. We reengineered the teleport application to give it a visual interface, using AR overlays in space and on the Active Bats.

Generally, our location-aware applications take over Bats as interaction devices. So, our aim was to design an interface that clearly shows users how our application interprets explicit (Bat) interaction. The application's detailed workings are as follows: Users press the bottom button to enable teleporting. If users move into a teleport region, they can use the top button to first connect to the server and then to scroll through their desktops on the server. If they move outside of the teleport region, pressing the top button will have no effect. If (at any time) they press the bottom button, teleporting will be disabled and all subsequent top button presses has no effect, regardless of whether users are inside or outside the teleport region. Even this small application presents a reasonable design problem. To keep the user informed, the interface must track and show two different kinds of state transitions in parallel: transitions in the domain model occurring as an effect of explicit interaction (top button pressed, bottom button pressed) and changes of context.

From the description, we see that the application's behavior is characterized by whether

- teleporting is on or off,
- the user is inside or outside a teleporting region, and
- the nearby computer is connected to the server.

Figure 3 shows how we specify the interaction in controllers. We assume that three models have been previously identified:

- the Bat model, which describes the Bat;
- the desktops model, which lists the user's desktops; and
- the teleport service model, which describes the teleport service state.

Because we're dealing with two contexts (inside and outside a teleport region), we had to design six controllers. The three possible actions—top button pressed, bottom button pressed, and context changed—map to controller functions **topBatButtonPressed**, **bottomBatButtonPressed**, and **activate** because a context change is always accompanied by controller activation. We use the **batMoved** function to update the Bat's position for the AR overlay. The system calls the functions in rows 1–3 when the corresponding explicit interaction event from the backend reaches the controller; it calls the functions **activate** and **deactivate** whenever the context that holds the particular controller is entered or left.

Each of the six controllers operates on one of three model-view pairs. Looking at cell B1 we see that when the user is in the teleport region and presses the top button with teleporting enabled, the controller initiates an action to connect the next desktop. This happens by effecting

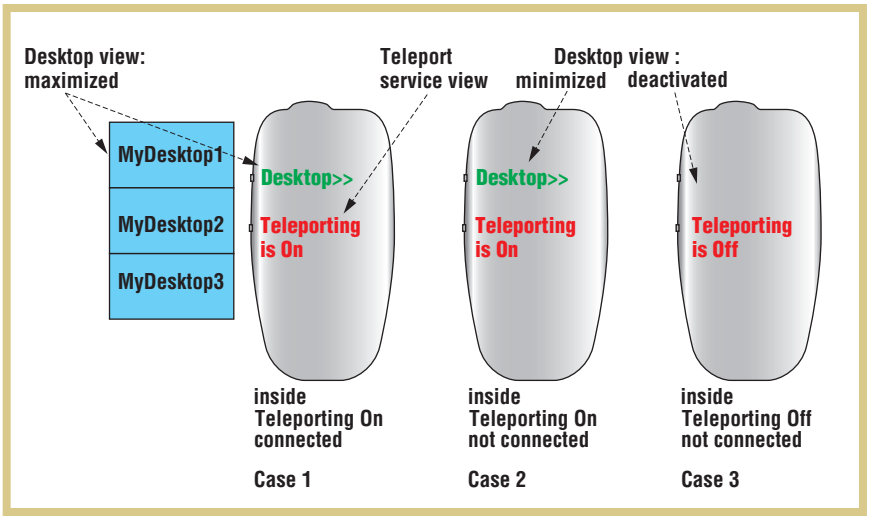


Figure 4. Three sets of views that users would see through AR goggles. The current context and explicit interaction state fully determine the look of the interface in each case.

a change in the desktops model; this eventually results in an update of the desktops view. The *desktops view* consists of a label for the button that controls the desktops ("Desktop>>") and a menu that lists the user's desktops.

From cell B4, we see that whenever the user enters the teleport region and is connected to a desktop, the system displays the menu and label. However, if the user has just entered the region and is not yet connected, the menu is not displayed (because the user has no desktops to choose from). If teleporting is inactive, the button isn't functional, so the system displays neither the button nor the menu. Figure 4 shows these maximized, minimized, and deactivated states. Finally, if the user leaves the teleport region, the system deactivates the view because teleporting is no longer available (cell B5).

When the Bat moves, the Bat controller updates the Bat model regardless of whether it is inside or outside the teleport region (cells A3 and D3). In our system, the *Bat view* is always a cuboid Bat overlay indicating where the system "sees" the Bat.

When a user presses the bottom button, the teleport service model updates regardless of whether the button press occurred inside or outside the teleport region (cells C2 and F2), eventually changing the view. The *teleport service view* displays a button label that shows whether the service is on or off.

The perhaps complicated design of the desktops view not only reflects the functionality available to users at any point, but can also be explained as follows: When there's a change of context, we must inform users in what explicit interaction state they are entering the new context. By looking at the interface, users can deduce the current context and interaction state.

Discussion

First and foremost, this is a frontend architecture. It therefore complements rather than competes with backend architectures that focus on the context acquisition and abstraction process, such as the Context Toolkit[10] or Spirit. Our architecture aims to construct a UI layer between users and the backend that communicates the system's inference process. We now contrast our architecture with how developers previously built applications using Spirit.

Architectural benefits

Before using our architecture, developers would build the teleport application in a monolithic manner. It was a piece of application logic that the backend would call if it sensed a user action. Then, for each possible case of action, the application would check for all possible combinations of conditions that could affect its response, such as, for example, whether the user was inside or outside a teleport region. This was done using nested IF statements. One problem is that this kind of cross-checking exponentially increases control flow complexity as the application is required to account for additional conditions.

Our architecture succeeds in reducing such condition-checking code by abstracting context-tracking and event-handling code common to all applications built within this framework. For example, it removes the need for the application developer to check what kind of event was received. This step is now performed in a "Spirit controller" superclass, resulting in a virtual function call of the event's handler. Similarly, developers no longer have to check for the current context for every action. Rather, they simply write a controller for each context and attach it to the corresponding context component, which ensures that the code piece runs only when the corresponding context is active. Developers still need to test domain-model-related state variables using IF statements, but each controller tests only variables that affect its model-view pair.

Our architecture's real power emerges when developers prototype UIs. To change how the application responds to events, developers need only edit corresponding controllers, rather than find all affected IF branches. We've essentially automated the process of finding the correct branch when an event arrives. So, what developers once had to tediously program now occurs at the superclass level; developers simply implement appropriate virtual functions to fit their individual applications.

However, our architecture's benefits over the traditional implementation exceed mere refactoring. It's comparable to traditional UI architectures such as MVC.[8] As such, it lets developers use object-oriented design to construct and communicate domain models. Furthermore, it provides a set of abstractions—

most notably the context component—to simplify the process of communicating the application's workings to the user.

Using the architecture

Our architecture underlies a toolkit we've used to build visually interactive location-aware applications. The toolkit itself requires system support,[5] including a ubicomp backend (such as Spirit), interface components between the backend and the application that tag events based on interaction type, and a rendering engine (which, in our system, is AR-based).

To build an application, you first model the domain using object-oriented analysis. Then, you implement models for each domain object making use of the facilities your ubicomp infrastructure provides you with. For each identified model, you write controllers and views. It might seem that writing so many controllers is a lot of work. But, as we showed in the monolithic version comparison, the controllers contain no more code than they'd need in the monolithic version anyway. In our case, we simply distribute that code among many more alternately activated components.

In developing this architecture, we wanted to show users how the application uses context. Existing toolkits for context-aware applications can take abstraction too far, architecturally separating the inference-managing components from the application itself.[9] Such components then make inferences without knowing how applications will use those inferences. Similarly, the applications don't know how the inferences were arrived at and thus can't communicate this to users. Our architecture successfully keeps the context tracking process within the application.

Further development

How could we further develop this architecture? Our architecture can account for physical information from various sources when it evaluates the context. However, we still must provide a context component for each context-variable combination to determine the contextual state. This can exponentially increase context components as the number of context variables grows. To counter this, we might be able to develop more complex—and possibly hierarchical—context components. However, the problem with introducing a hierarchical organization is that the context that people create and maintain in daily interactions isn't hierarchical.

Perhaps a greater limiting factor on the number of possible context variables is the number of dependencies users can actually understand. Also, we're not trying to model all information that the system can sense about the world (the backend can manage that). Our goal is to model only the context relevant to a particular application's interactions.

Another limit of our framework is that it assumes only one user. This is actually a limitation of our interaction model: we've looked at conversation as a one-to-one activity. We are planning to research how context is established and maintained in real-world multiparty conversations so we can model it accordingly.

We have derived our framework from an interaction model that was inspired by a particular sociological view of what context means for people in everyday life. So, your decision about whether to use the architecture must be partially based on how much of this view you accept. We believe our architecture is beneficial if you

- believe that context is a property of interaction, rather than objects or people,[7]
- want your applications to model this view, and
- want to visually convey to users how the application is using context.

If you believe that context is synonymous with physical information, you'll have difficulty applying this architecture. Applications such as electronic reminders that trigger only when users enter particular locations are called "context-aware" on this basis. To exploit our architecture's full potential, however, your application must contain a large interaction component and adapt its look and feel to different contexts. After all, it's an architecture for *interactive* context-aware applications. P

> In a conversation, context isn't a piece of information about a person or an object: it's a frame of reference for interpreting content.

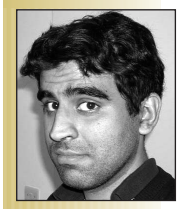
ACKNOWLEDGMENTS

We are grateful to Andy Hopper for providing vision and support throughout this research project.

REFERENCES

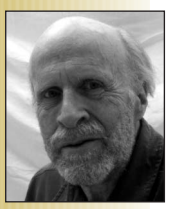
1. K. Rehman, F. Stajano, and G. Coulouris, "Interfacing with the Invisible Computer," *Proc. 2nd Nordic Conf. Human-Computer Interaction* (NordiCHI), ACM Press, 2002, pp. 213–216.

the AUTHORS



Kasim Rehman is an associate at Cambridge Systems Associates, where he works on financial software. His research interests include interaction design and visualization. He received his PhD in ubiquitous computing from the University of Cambridge. Contact him at Cambridge Systems Associates, 5-7 Portugal Place, Cambridge, UK; kasim_rehman@cantab.net.

Frank Stajano is a tenured lecturer (associate professor) at the University of Cambridge Computer Laboratory. His research interests include computer security, ubiquitous computing, and privacy in the electronic society. He received his PhD in computer security from the University of Cambridge. He is the author of *Security for Ubiquitous Computing* (Wiley, 2002). Contact him at the Univ. of Cambridge Computer Laboratory, 15 J.J. Thomson Ave., Cambridge CB3 0FD, UK; frank.stajano@cl.cam.ac.uk; www.cl.cam.ac.uk/~fms27.



George Coulouris is an emeritus professor of computer systems at the University of London and a senior visiting fellow in the University of Cambridge Computer Laboratory. His research interests include ubiquitous and sentient systems. He received a BSc in physics at University College London. Contact him at the Digital Technology Group, Computer Laboratory, Cambridge Univ., William Gates Building, Cambridge CB3 0FD, UK: gfc22@cam.ac.uk.

2. V. Bellotti et al., "Making Sense of Sensing Systems: Five Questions for Designers and Researchers," *Proc. Computer-Human Interaction* (CHI), ACM Press, 2002, pp. 415–422.

3. M. Weiser and J.S. Brown, "The Coming Age of Calm Technology" *Xerox PARC*, 5 Oct. 1996; www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm.

4. D.A. Norman, *The Design of Everyday Things*, MIT Press, 1989.

5. K. Rehman, F. Stajano, and G. Coulouris, "Visually Interactive Location-Aware Computing," *Proc. 7th Int'l Conf. Ubiquitous Computing* (Ubicomp), LNCS 3660, Springer, 2005, pp. 177–194.

6. A. Harter et al., "The Anatomy of a Context-Aware Application," *Proc. 5th Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking* (Mobicom), ACM Press, 1999, pp. 59–68.

7. P. Dourish, "What We Talk About When We Talk About Context," *Personal and Ubiquitous Computing*, vol. 8, no. 1, 2004, pp. 19–30.

8. G. Krasner and S. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *J. Object Oriented Programming*, vol. 1, no. 3, 1988, pp. 26–49.

9. V. Bellotti and K. Edwards, "Intelligibility and Accountability: Human Considerations in Context-Aware Systems," *Human-Computer Interaction*, vol. 16, nos. 2–4, 2001, pp. 193–212.

10. A. Dey, G.D. Abowd, and D. Salber, "A Context-Based Infrastructure for Smart Environments," *Managing Interactions in Smart Environments* (MANSE 99), ACM Press, 1999, pp. 114–128.