Human–Computer Interaction

ISSN: 0737-0024 (Print) 1532-7051 (Online) Journal homepage: https://www.tandfonline.com/loi/hhci20

A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications

Anind K. Dey , Gregory D. Abowd & Daniel Salber

To cite this article: Anind K. Dey , Gregory D. Abowd & Daniel Salber (2001) A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, Human–Computer Interaction, 16:2-4, 97-166, DOI: 10.1207/S15327051HCI16234_02

To link to this article: https://doi.org/10.1207/S15327051HCI16234_02

Published online: 12 Nov 2009.

Submit your article to this journal

Article views: 1475

Citing articles: 1242 View citing articles

HUMAN-COMPUTER INTERACTION, 2001, Volume 16, pp. 97–166

A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications

Anind K. Dey and **Gregory D. Abowd**
Georgia Institute of Technology

Daniel Salber
IBM T.J. Watson Research Center

ABSTRACT

Computing devices and applications are now used beyond the desktop, in diverse environments, and this trend toward ubiquitous computing is accelerating. One challenge that remains in this emerging research field is the ability to enhance the behavior of any application by informing it of the *context* of its use. By context, we refer to any information that characterizes a situation related to the interaction between humans, applications, and the surrounding environment. Context-aware applications promise richer and easier interaction, but the

Anind Dey is a computer scientist with an interest in context-aware and ubiquitous computing; a recent Ph.D., he is a Senior Researcher with Intel Research in Berkeley, California. **Gregory Abowd** is a computer scientist with interests in HCI and software engineering challenges associated with ubiquitous computing; he is an Associate Professor in the College of Computing and GVU Center at Georgia Tech. **Daniel Salber** is a computer scientist with an interest in software engineering for context aware applications; he is a Research Associate in the Pervasive Computing Applications group at the IBM T.J. Watson Research Center.

CONTENTS

current state of research in this field is still far removed from that vision. This is due to 3 main problems: (a) the notion of context is still ill defined, (b) there is a lack of conceptual models and methods to help drive the design of context-aware applications, and (c) no tools are available to jump-start the development of context-aware applications. In this anchor article, we address these 3 problems in turn. We first define context, identify categories of contextual information, and characterize context-aware application behavior. Though the full impact of context-aware computing requires understanding very subtle and high-level notions of context, we are focusing our efforts on the pieces of context that can be inferred automatically from sensors in a physical environment. We then present a conceptual framework that separates the acquisition and representation of context from the delivery and reaction to context by a context-aware application. We have built a toolkit, the Context Toolkit, that instantiates this conceptual framework and supports the rapid development of a rich space of context-aware applications. We illustrate the usefulness of the con-

ceptual framework by describing a number of context-aware applications that have been prototyped using the Context Toolkit. We also demonstrate how such a framework can support the investigation of important research challenges in the area of context-aware computing.

1. INTRODUCTION

The typical user is not facing a desktop machine in the relatively predictable office environment anymore. Rather, users have to deal with diverse devices (mobile or fixed) sporting diverse interfaces and used in diverse environments. In appearance, this phenomenon is a step toward the realization of Weiser's (1991) ubiquitous computing paradigm, or "third wave of computing," where specialized devices outnumber users (Weiser, 1991). However, many important pieces necessary to achieve the ubiquitous computing vision are not yet in place. Most notably, interaction paradigms with today's devices fail to account for a major difference with the static desktop interaction model. Devices are now often used in changing environments, yet they do not adapt to those changes very well. Although moving away from the desktop brings up a new variety of situations in which an application may be used, computing devices are left unaware of their surrounding environment. One hypothesis that a number of ubiquitous computing researchers share is that enabling devices and applications to automatically adapt to changes in their surrounding physical and electronic environments will lead to an enhancement of the user experience.

The information in the physical and electronic environments creates a *context* for the interaction between humans and computational services. We define context as any information that characterizes a situation related to the interaction between users, applications, and the surrounding environment. A growing research activity within ubiquitous computing deals with the challenges of *context-aware computing*. Though the notion of context can entail very subtle and high-level interpretations of a situation, much of the effort within the ubiquitous computing community takes a bottom-up approach to context. The focus is mainly on understanding and handling context that can be sensed automatically in a physical environment and treated as implicit input to positively affect the behavior of an application.

Apart from the demonstration of a variety of prototype context-aware applications, the majority of which are location-based services, there has been relatively little advancement in context-aware computing over the past 5 years. There are technology- and human-centered challenges that stem from a poor understanding of what constitutes context and how it should be represented. We lack conceptual models and tools to support the rapid development of rich context-aware applications that might better inform the empirical investigation of interaction design and the social implications of context-aware computing. The work presented here attempts to enable a new phase of context-aware applications development. We want to help applications developers understand what context is, what it can be used for, and provide concepts and practical support for the software design and construction of context-aware applications.

1.1. The Conference Assistant Scenario

To give the reader an idea of the richness of context-aware computing we are initially aiming for, we present a scenario of an automated service meant to support attendees at an academic conference:

> Ashley is attending a technical conference. The conference features a large number of presentations and demos spread over multiple tracks. Ashley is attending the conference with her colleagues Bob and John and they have decided to try to attend different presentations and to share what they have learned at the end of the day. When Ashley picks up her conference package on the first day, she provides her contact information and the topics she's most interested in. She also mentions that her colleagues Bob and John are attending. Along with the conference proceedings, she receives a personal conference assistant, a handheld device designed to guide and assist her throughout the conference.
>
> Ashley has a hard time deciding what to attend for the first session. The sessions start in 5 min. She turns to the conference assistant. Based on her interests, it recommends a presentation and a demo and gives her directions to get to the presentation and demo rooms. Ashley chooses the presentation and heads to the room. When she enters the presentation room, the conference assistant displays information about the session that is about to begin, including information on the presenter, relevant URLs, and the proceedings' page numbers of the papers being presented. It also shows a thumbnail of the current slide and offers note-taking capabilities.
>
> During the presentation, Ashley uses the conference assistant note-taking feature to jot down some comments. The conference assistant indicates that both audio and video, as well as the slides' content, are recorded for this session. She marks the diagrams that interest her for later retrieval. At the end of the presentation, she addresses a question to the speaker. Her question relates to a particular detail of a diagram on one of the speaker's slides. Using the conference assistant, she

is able to bring up the relevant slide on the room's screen to make her question clearer to the speaker and the audience. The last talk of the session is cancelled because the speaker missed his flight. Ashley wonders what to do and looks to the conference assistant to find out what Bob and John are up to. The conference assistant shows her that they are both attending different sessions. The conference assistant reports that Bob has indicated that his session is moderately interesting. John however, seems to be enjoying his session a lot. Ashley decides to join John's session.

Back at work after the conference, Ashley connects to the conference Web site to refresh her memory and write up her trip report. She logs in at the conference Web site. It shows her the talks she attended and allows her to browse the notes she took. She can also watch and listen to the recorded sessions she has missed. Her trip report contains complete coverage of the sessions of the conference that are relevant to her team and it is written in a snap!

We have implemented a scaled-down version of the conference assistant application intended to support visitor visits during our lab's demo days. Our application supports many features described in this scenario. In Section 8 of this anchor article, we revisit the conference assistant and describe in detail our design and architecture. But first, we explain why this type of application is difficult to build today, and we introduce a number of concepts that make context-aware computing easier to comprehend for application designers and developers.

1.2. The Difficulties of Handling Context

The Conference Assistant scenario outlines a variety of possible uses of context information to enhance interaction with devices in dynamic settings. Prior to our work, there has not been a context-aware application that approaches this complexity, in terms of the variety of context used and the ways in which it is used. Difficulties arise in the design, development, and evolution of context-aware applications. Designers lack conceptual tools and methods to account for context awareness. As a result, the choice of context information used in applications is very often driven by the context acquisition mechanisms available—hardware and software sensors. This approach entails a number of risks. First, the initial choice of sensors may not be the most appropriate. The details and shortcomings of the sensors may be carried up to the application level and hinder the flexibility of the interaction and further evolution of the application. More important, a sensor-driven approach constrains the possibilities of designers by limiting the kind of applications they are able to design and the context uses they can imagine.

Developers face another set of problems related to distribution, modifiability, and reusability. Context-aware applications are often distributed because they acquire or provide context information in a number of different places. For example, in the Conference Assistant scenario, each room and venue of the conference must be able to sense the presence and identity of users. Although mechanisms for distribution are now mainstream on desktops and servers, they are not always appropriate for distributed networks of sensors. Indeed, context awareness is most relevant when the environment is highly dynamic, such as when the user is mobile. Thus, context-aware applications may be implemented on very diverse kinds of computing platforms, ranging from handheld devices to wearable computers to custom-built embedded systems. As a result, context-aware applications require lightweight, portable, and interoperable mechanisms across a wide range of platforms.

As with graphical user interfaces (GUIs), and more crucially so given the overall lack of experience with context-aware applications, iterative development is key to creating usable context-aware applications. Thus, applications must be implemented in a way that makes it easy to modify context-related functions. There are currently few guidelines, models, or tools that support this requirement. Finally, application developers should be able to reuse satisfactory context-aware solutions. But no methods or tools make this task easy either.

1.3. Overview of Anchor Article

This anchor article has three main goals: (a) to provide an operational understanding of context from which we can derive concepts useful for context-aware applications development, (b) to introduce a conceptual framework to assist in the design of context-aware applications, and (c) to present a toolkit that can facilitate context-aware computing research by allowing the empirical investigation of the design space and the exploration of difficult challenges in the handling of implicitly sensed context. In Section 2, we present an overview of the current understanding and uses of context in the context-awareness literature. We propose practical basic context types and context uses that help clarify the design space for context-aware computing. In Section 3, we present requirements for a framework that supports the acquisition, representation, delivery, and reaction to context information that can be automatically sensed and used as implicit input to affect application behavior. In Section 4 we discuss relevant details on the Context Toolkit, an instantiation of the conceptual framework (Dey, 2000).[1] In Sections 5 through

1. Further details on the Context Toolkit, including downloadable versions and a tutorial, can be found at http://www.cc.gatech.edu/fce/contexttoolkit

9 we explore some of the design space of context-aware applications with a description of various applications we have developed using the Context Toolkit. The final example is a detailed discussion of the design and implementation of a prototype of the Conference Assistant. In Section 10, we discuss some related work in conceptual frameworks and architectures to support context-aware computing and compare that work to what has been presented in this anchor article. In Section 11, we demonstrate how some existing challenges in context-aware computing—better representations of context, handling higher level notions of context, support for privacy concerns, support for imperfect or ambiguous sensed context, and higher level programming abstractions—can all be explored as result of the conceptual framework and Context Toolkit.

2. UNDERSTANDING AND USING CONTEXT

In this section, we use a survey of context-aware computing performed by Anind K. Dey and Gregory D. Abowd (2000b) to provide a better understanding of what context is and how it can be used. First, we look at previous definitions of context in the ubiquitous computing literature. We build on these definitions to provide a general definition of context that we then refine by looking at entities that provide context information. We next define context awareness and identify three key categories of context-aware functions.

2.1. Defining Context

We would like to provide an operational definition of context that helps determine exactly what it is. This will enable us to establish what the common characteristics of context are, which can be supported by abstractions and tools. According to *Webster's New Twentieth Century Dictionary* (1980), context is the "whole situation, background or environment relevant to some happening or personality." This definition is too general to be useful in context-aware computing. Definitions of context by previous authors fall in two categories: enumerations of examples of context information and categorizations.

Previous Definitions

In the work that first defined the term *context-aware,* Schilit and Theimer (1994) referred to context as location, identities of nearby people and objects and changes to those objects. In a similar definition, Brown, Bovey, and Chen (1997) defined context as location, identities of the people around the user, the time of day, season, temperature, and so forth. Ryan, Pascoe, and Morse (1997) defined context as the user's location, environment, identity, and time.

Dey (1998) enumerated context as the user's emotional state, focus of attention, location and orientation, date and time, and objects and people in the user's environment. These definitions that define context by example are difficult to apply. When considering a potential new type of context information, it is not clear how the definition can help us decide whether to classify the information as context. For example, none of the previous definitions helps decide whether a user's preferences or interests are context information.

Other definitions have provided synonyms for context, referring to context as the environment or situation. Some consider context to be the user's environment, whereas others consider it to be the application's environment. For example, Brown (1996) defined context to be the elements of the user's environment that the computer knows about. Franklin and Flaschbart (1998) saw it as the situation of the user. Ward, Jones, and Hopper (1997) viewed context as the state of the application's surroundings and Rodden, Cheverst, Davies, and Dix (1998) defined it as the application's setting. Hull, Neaves, and Bedford-Roberts (1997) included the entire environment by defining context to be aspects of the current situation. These definitions are clearly more general than enumerations, but this generality is also a limitation. These latter definitions provide little guidance to analyze the constituent elements of context, much less identify them.

The definitions by Schilit, Adams, and Want (1994); Dey, Abowd, and Wood (1998); and Pascoe (1998) are closest in spirit to the operational definition we seek. Schilit et al. (1994) claimed that the important aspects of context are: where the user is, who the user is with, and what resources are nearby. They define context to be the constantly changing execution environment. The environment is threefold:

- Computing environment: available processors, devices accessible for user input and display, network capacity, connectivity, and costs of computing.
- User environment: location, collection of nearby people, and social situation.
- Physical environment: lighting and noise level.

Dey et al. (1998) defined context as the user's physical, social, emotional, or informational state. Finally, Pascoe (1998) defined context to be the subset of physical and conceptual states of interest to a particular entity.

Our Definition

Based on these prior attempts to define context, we proceed with the following definition, taken from our previous work (Dey & Abowd, 2000b):

> **Context:** any information that can be used to characterize the situation of entities (i.e., whether a person, place, or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity, and state of people, groups, and computational and physical objects.

This definition encompasses the definitions given by previous authors. As stated earlier, a goal of context acquisition is to determine what a user is trying to accomplish. Because the user's objective is difficult to determine directly, context cues can be used to help infer this information and to inform an application on how to best support the user. The definition we have provided is quite general. This is because context awareness represents a generalized model of input, including both implicit and explicit input, allowing almost *any* application to be considered more or less context aware insofar as it reacts to input. In this anchor article, we concentrate on the gathering and use of implicit environmental-sensed input by applications. The conceptual framework we present can be used for both explicit and implicit input, but focuses on supporting the ease of incorporating implicit input into applications.

Our definition does not differentiate between manually acquired information and automatically acquired information. There is a great division among context-aware researchers as to whether context should only include automatically acquired information, or both manually and automatically acquired information. We have opted for a more inclusive definition. In an ideal setting, context would be obtained automatically and there would be no need for manual acquisition. However, in the real world, most context cannot be sensed automatically and applications must rely on the user to manually provide it. If information is relevant to an interaction, then how that information is acquired should not impact whether it is seen as context or not. The main goal for providing this definition of context is to offer guidance in identifying broad categories of context.

2.2. Categories of Context

Looking at the scenario of Section 1.1, the Conference Assistant needs to acquire various kinds of information about the environment to perform correctly: the location of the user, the topics that might interest the user, the contents of the presentation, whether audio or video are recorded for the session, and the location of other users. Given the diversity of context information, it is useful to attempt to categorize it to make it easier to comprehend in a systematic manner. We, therefore, introduce a simple classification of context information based on the entities whose context is assessed and categories of context information.

The entities we identified as most relevant are places, people, and things. *Places* are regions of geographical space such as rooms, offices, buildings, or streets. *People* can be either individuals or groups, co-located or distributed. *Things* are either physical objects or software components and artifacts (e.g., an application or file).

We introduce four essential categories, or characteristics, of context information—identity, location, status (or activity), and time. *Identity* refers to the ability to assign a unique identifier to an entity. The identifier has to be unique in the namespace that is used by the applications. *Location* is more than just position information in a two-dimensional space. It is expanded to include orientation and elevation, as well as all information that can be used to deduce spatial relations between entities, such as co-location, proximity, or containment. For example, the information that an object is upside down is location information. Location also applies to places. Places can be located in a frame of reference such as geographical coordinates or relative spatial relations, and thus have a location. *Status* (or *activity*) identifies intrinsic characteristics of the entity that can be sensed. For a place, it can be the current temperature or the ambient light or noise level. For a person, it can refer to physiological factors such as vital signs or tiredness, or the activity the person is involved in, such as reading or talking. Similarly, for a group of people, status is a characteristic of the group such as their enthusiasm or global mood, or a description of their activity, such as attending a lecture or having a meeting. For software components, status basically refers to any attribute of the software component that can be queried. Typical examples are the uptime or load of a CPU, the existence or state of files in a file system, or the state of a software component such as an application. Finally, *time* is context information as it helps characterize a situation. It enables us to leverage off the richness and value of historical information. It is most often used in conjunction with other pieces of context, either as a timestamp or as a time span, indicating an instant or period during which some other contextual information is known or relevant. However, in some cases, just knowing the relative ordering of events or causality is sufficient.

The reason for our choice of these basic context categories is that the information they provide can be used to infer additional pieces of context and lead to a more extensive assessment of a situation. Simple *inference*, or *derivation,* of context information happens when related context information is deduced from a single known piece of context information. For example, the phone number or the address of a person can be derived from her identity. More complex inference of context information refers to the process of considering several pieces of context information that describe a situation, and using them to infer a new piece of context information. For example, by knowing that a room is occupied, the number of people in the room, their relative positions in the room, and if they are talking or not, one can determine if a meeting is taking place.

The context categories we have identified, as well as inferred context information, can be used in a number of ways by context-aware applications. To further clarify context and its uses, we now turn to functions that make an application context aware.

2.3. Context-Aware Functions

Having presented a definition of context and identified key context types, we must now examine how applications can effectively use context information. In our survey of context-aware computing, we proposed a classification of context-aware functions that a context-aware application may implement (Dey & Abowd, 2000b). This classification introduces three categories of functions, related to the presentation of information, the execution of services, and the storage of context information attached to other captured information for later retrieval.

The first category, *presenting information and services*, refers to applications that either present context information to the user or use context to propose appropriate selections of actions to the user. There are several examples of this class of functions in the literature and in commercially available systems: showing the user's or her vehicle's location on a map and possibly indicating nearby sites of interest (Abowd et al., 1997; Bederson, 1995; Davies, Mitchell, Cheverst, & Blair, 1998; Feiner, MacIntyre, Hollerer, & Webster, 1997; Fels et al., 1998; McCarthy & Anagost, 2000; McCarthy & Meidel, 1999); presenting a choice of printers close to the user (Schilit et al., 1994); sensing and presenting in/out information for a group of users (Salber, Dey, & Abowd, 1999); ambient information displays (Heiner, Hudson, & Tanaka, 1999; Ishii & Ullmer, 1997; Mynatt, Back, Want, Baer, & Ellis, 1998; Weiser & Brown, 1997); and providing remote awareness of others (Schmidt, Takaluoma, & Mäntyjärvi, 2000).

The second category, *automatically executing a service*, describes applications that trigger a command or reconfigure the system on behalf of the user according to context changes. Examples include: the Teleport system in which a user's desktop environment follows her as she moves from workstation to workstation (Want, Hopper, Falcao, & Gibbons, 1992); car navigation systems that recompute driving directions when the user misses a turn (Hertz, 1999); a recording whiteboard that senses when an informal and unscheduled encounter of individuals occurs and automatically starts recording the ensuing meeting (Brotherton, Abowd, & Truong, 1999); mobile devices enhanced with sensors that determine their context of use to change their settings and actions

(Harrison, Fishkin, Gujar, Mochon, & Want, 1998; Hinckley, Pierce, Sinclair, & Horvitz, 2000; Schmidt et al., 1999); a camera that captures an image when the user is startled as sensed by biometric sensors (Healey & Picard, 1998); and devices that deliver reminders when users are at a specified location (Beigl, 2000; Marmasse & Schmandt, 2000).

In the third category, *attaching context information for later retrieval*, applications tag captured data with relevant context information. For example, a zoology application tags notes taken by the user with the location and the time of the observation (Pascoe, Ryan, & Morse, 1998). The informal meeting capture system mentioned earlier provides an interface to access informal meeting notes based on who was there, when the meeting occurred, and where the meeting was located. In a similar vein are Time-Machine Computing (Rekimoto, 1999) and Placeless Documents (Dourish et al., 2000), two systems that attach context to desktop or networked files to enable easier retrieval. Some of the more complex examples in this category are memory augmentation applications such as Forget-Me-Not (Lamming & Flynn, 1994) and the Remembrance Agent (Rhodes, 1997).

In this section, we have established a definition of context and we have further elicited the dimensions of context information, as well as its possible uses in context-aware applications. We now turn to the software design of context-aware applications.

3. REQUIREMENTS AND CONCEPTUAL FRAMEWORK FOR HANDLING CONTEXT

The software design of context-aware applications raises new challenges. Handling context is difficult for at least three reasons: (a) there are no guiding principles to support good software engineering practices, (b) designers lack abstractions to think about context, and (c) context sensing is very often distributed and leads to complex distributed designs. In this section, we use separation of concerns between context acquisition and the use of context in applications as a guiding principle. We derive abstractions that help acquire, collect, and manage context in an application-independent fashion and identify corresponding software components. These components, along with a simple distributed platform, form the basis of our Context Toolkit, described in Section 4.

When looking at a new technological advance and when trying to make it easily accessible to software designers, a common stepping stone is to apply tried-and-true software engineering principles to the particular case of the new technology at hand. With the advent of GUIs, the separation of concerns between the actual application (often at first a legacy text-based application) and the new world of windows, icons, menus, and pointers quickly became paramount. This requirement led to software architecture models, design processes, and development tools to more effectively support the design and development of GUIs. Other principles, such as reusability or modifiability, stemmed from the particular nature of GUI design and development, where original solutions are costly to develop and iterative design is key to usability.

With context-aware applications, we are faced with a similar problem. Context awareness opens new possibilities that can drive the development of new applications but can also be beneficial to legacy applications. Whereas GUI-based applications focus on explicit input, context-aware applications try to take a more general notion of input, focusing on implicit input but dealing with explicit input as well. Context-aware applications need to take advantage of the sensors and sensing techniques available. Thus, as with GUIs, our major preoccupation is to achieve a separation between the application per se and context acquisition and processing.

Inspired by concepts commonly found in GUI toolkits, we have defined some abstractions that help infer higher level information from context and support separation of concerns. We have also identified requirements for dealing with context that are extensions of traditional GUI capabilities. In this section, we introduce these requirements and the context abstractions we have defined, and describe how to derive the abstractions from the application's specifications. The abstractions form a conceptual framework that can be used to design context-aware applications.

3.1. Requirements for Dealing With Context

With the perspective of an application designer in mind, we wanted to provide a conceptual framework that automatically supports all the tasks that are common across applications, requiring the designer to only provide support for the application-specific tasks. To this end, we have identified a number of requirements that the framework must fulfill to enable designers to more easily deal with context. These requirements are:

- Separation of concerns.
- Context interpretation.
- Transparent, distributed communications.
- Constant availability of context acquisition.
- Context storage and history.
- Resource discovery.

We now discuss each of these requirements.

Separation of Concerns

One of the main reasons why context is not used more often in applications is that there is no common way to acquire and handle context. In general, context is handled in an improvised fashion. Application developers choose whichever technique is easiest to implement, at the expense of generality and reuse. We now look at two common ways in which context has been handled: connecting sensor drivers directly into applications and using servers to hide sensor details.

With some applications (Harrison et al., 1998; Rekimoto, 1996), the drivers for sensors used to detect context are directly hardwired into the applications themselves. In this situation, application designers are forced to write code that deals with the sensor details, using whatever protocol the sensors dictate. There are two problems with this technique. The first problem is that it makes the task of building a context-aware application very burdensome by requiring application builders to deal with the potentially complex acquisition of context. The second problem with this technique is that it does not support good software engineering practices. The technique does not enforce separation between application semantics and the low-level details of context acquisition from individual sensors. This leads to a loss of generality, making the sensors difficult to reuse in other applications and difficult to use simultaneously in multiple applications.

Ideally, we would like to handle context in the same manner as we handle user input. User interface toolkits support application designers in handling input. They provide an important abstraction to enable designers to use input without worrying about how the input was collected. This abstraction is called a *widget,* or an *interactor.* The widget abstraction provides many benefits and has been used not only in standard keyboard and mouse interaction, but also with pen and speech input (Arons, 1991) and with the unconventional input devices used in virtual reality (MacIntyre & Feiner, 1996). It facilitates the separation of application semantics from low-level input handling details. For example, an application does not have to be modified if a pen is used for pointing rather than a mouse. It supports reuse by allowing multiple applications to create their own instances of a widget. It contains not only a querying mechanism but also possesses a notification, or *callback*, mechanism to allow applications to obtain input information as it occurs. Finally, in a given toolkit, all the widgets have a common external interface. This means that an application can treat all widgets in a similar fashion, not having to deal with differences between individual widgets.

There are systems that support event management (Bauer, Heiber, Kortuem, & Segall, 1998; Schilit, 1995), either through the use of querying mechanisms, notification mechanisms, or both, to acquire context from sensors. However, this previous work has suffered from the design of specialized servers, which do not share a common interface (Bauer et al., 1998). This forces an application to deal with each server in a distinct manner, rather than being able to deal with all servers the same way. On the server implementation side, with no common support, it is more difficult to produce servers, resulting in a minimal range of server types being used (e.g., focusing on location; Schilit, 1995). It is not a requirement that both querying and notification mechanisms be supported because one can be used to implement the other. For reasons of flexibility, it is to an application's advantage that both be available (Rodden et al., 1998). Querying a sensor for context is appropriate for one-time context needs. But the sole use of querying requires that applications be proactive (by polling) when requesting context information from sensors. Once it receives the context, the application must then determine whether the context has changed and whether those changes are interesting or useful to it. The notification or publish/subscribe mechanism is appropriate for repetitive context needs, where an application may want to set conditions on when it wants to be notified.

By separating how context is acquired from how it is used, applications can now use contextual information without worrying about the details of a sensor and how to acquire context from it. These details are not completely hidden and can be obtained if needed.

Context Interpretation

There is a need to extend the notification and querying mechanisms to allow applications to retrieve context from distributed computers. There may be multiple layers that context data go through before reaching an application, due to the need for additional abstraction. This can be as simple as needing to abstract a smart card id into its owner's name, but can also be much more complex. For example, an application wants to be notified when meetings occur. At the lowest level, location information can be interpreted to determine where various users are and identity information can be used to check co-location. At the next level, this information could be combined with sound level information and scheduling information to determine if a meeting is taking place. From an application designer's perspective, the use of these multiple layers should be transparent. To support this transparency, context must often be interpreted before it can be used by an application. An application may not be interested in the low-level information, and may only want to know when a meeting starts. For the interpretation to be easily reusable by multiple applications, it needs to be provided by the architecture. Otherwise, each application would have to reimplement the necessary implementation. There are a number of systems that provide mechanisms to perform transparent recursive interpretation (Dey et al., 1998; Kiciman & Fox, 2000).

Transparent, Distributed Communications

Traditional user input comes from the keyboard and mouse. These devices are connected directly to the computer with which they are being used. When dealing with context, the devices used to sense context most likely are not attached to the same computer running an application that will react to that context. For example, an indoor infrared positioning system may consist of many infrared emitters and detectors in a building. The sensors might be physically distributed and cannot all be directly connected to a single machine. In addition, multiple applications may require use of that location information and these applications may run on multiple computing devices. As environments and computers are becoming more instrumented, more context can be sensed, but this context will be coming from multiple, distributed machines connected via a computer network. The fact that communication is distributed should be transparent to both sensors and applications. This simplifies the design and building of both sensors and applications, relieving the designer of having to build a communications framework. Without it, the designer would have to design and implement a communications protocol and design and implement an encoding scheme (and accompanying decoder) for passing context information.

A related requirement is the need for a global timeclock mechanism. Typically, when dealing with a distributed computing environment, each of the distributed computers maintain their own clock and are not synchronized with each other. From our earlier discussion of context (Section 2.1), we see that time is a very important type of context. To accurately compare and combine context arriving from distributed computers, these computers must share the same notion of time and be synchronized to the greatest extent possible.

Constant Availability of Context Acquisition

With GUI applications, user interface components such as buttons and menus are instantiated, controlled, and used by only a single application (with the exception of some groupware applications). In contrast, context-aware applications should not instantiate individual components that provide sensor data, but must be able to access existing ones, when they require it. Furthermore, multiple applications may need to access the same piece of context. This leads to a requirement that the components that acquire context must be executing independently from the applications that use them. This eases the programming burden on the application designer by not requiring her to instantiate, maintain, or keep track of components that acquire context, while allowing her to easily communicate with them.

Because these components run independently of applications, there is a need for them to be persistent and available all the time. It is not known a priori when applications will require certain context information; consequently, the components must be running perpetually to allow applications to contact them when needed. Context is information that should always be available. Take the call-forwarding example from the Active Badge research (Want et al., 1992). When a phone call was received, an application tried to forward the call to the phone nearest the intended recipient. The application could not locate the user if the Badge server was not active. If the Badge server were instantiated and controlled by a single application, other applications could not use the context it provides.

Context Storage and History

A requirement linked to the need for constant availability is the desire to maintain historical information. User input widgets maintain little, if any, historical information. For example, a file selection dialog box keeps track of only the most recent files that have been selected and allows a user to select those easily. In general though, if a more complete history is required, it is left up to the application to maintain it. In contrast, a component that acquires context information should maintain a history of all the context it obtains. Context history can be used to establish trends and predict future context values. Without context storage, this type of analysis could not be performed. A component may collect context when no applications are interested in that particular context information. Therefore, there may be no applications available to store that context. However, there may be an application in the future that requires the history of that context. For example, an application may need the location history for a user to predict his future location. For these reasons, the architecture must support the storage of context. Ideally, the architecture would support storage at the finest level of detail possible to meet any application's requests for historical information.

Resource Discovery

For an application to communicate with a sensor (or rather its software interface), it must know what kind of information the sensor can provide, where it is located, and how to communicate with it (protocol, language, and mechanisms to use). For distributed sensors, this means knowing, at a minimum, both the hostname and port of the computer the sensor is running on. To be able to effectively hide these details from the application, the architecture needs to support a form of resource discovery (Schwartz, Emtage, Kahle, & Neuman, 1992). With a resource discovery mechanism, when an application is started, it could specify the type of context information required. The mechanism would be responsible for finding any applicable components and for providing the application with ways to access them. For example, in the case of a simple In/Out Board application, rather than hardcoding the location of the sensing technology being used, the developer can indicate that the application is to be notified whenever any user of the In/Out Board enters or leaves the building. This information can then be provided by any source of location context.

3.2. Context Abstractions

We now describe a conceptual framework that meets the requirements presented in the previous section. Roughly speaking, we present a framework that matches the physical world of sensors and supports the ability to transform and collect contextual information. After discussing the framework, we demonstrate how it can be used to build context-aware applications.

A context widget is a software component that provides applications with access to context information from their operating environment. In the same way that GUI widgets mediate between the application and the user, context widgets mediate between the application and its operating environment. As a result, just as GUI widgets insulate applications from some presentation concerns, context widgets insulate applications from context acquisition concerns. To address context-specific operations, we introduce four additional categories of components in our conceptual framework: interpreters, aggregators, services, and discoverers.

Context Widgets

GUI widgets (a) hide the specifics of the input devices being used from the application programmer, allowing changes with minimal impact on applications; (b) manage interaction to provide applications with relevant results of user actions; and (c) provide reusable building blocks. Similarly, context widgets provide the following benefits:

- They provide a separation of concerns by *hiding the complexity* of the actual sensors used from the application. Whether the location of a user is sensed using Active Badges, floor sensors, a radio frequency- (RF-) based indoor positioning system, or a combination of these should not impact the application.

- They *abstract context information* to suit the expected needs of applications. A widget that tracks the location of a user within a building or a city notifies the application only when the user moves from one room to another, or from one street corner to another, and doesn't report less significant moves to the application. Widgets provide abstracted information that we expect applications to need the most frequently.
- They *provide reusable and customizable building blocks* of context sensing. A widget that tracks the location of a user can be used by a variety of applications, from tour guides to car navigation to office awareness systems. Furthermore, context widgets can be tailored and combined in ways similar to GUI widgets. For example, a presence widget senses the presence of people in a room. A meeting widget may be built on top of a presence widget and assume a meeting is beginning when two or more people are present.

The widget abstraction has been criticized in the GUI arena because it hides the underlying details of how input is being collected. Although context widgets support a uniform interface to allow applications to acquire context, they do not completely hide the underlying details of how context is being acquired from sensors. This includes details on the type of sensor used, how data are acquired from the sensor, and the resolution and accuracy of the sensor. The default behavior is to not provide these details to applications. However, applications can request this information, if desired.

From the application's perspective, context widgets encapsulate context information and provide methods to access it in a way very similar to a GUI widget. Context widgets provide callbacks to *notify* applications of significant context changes and attributes that can be *queried* or *polled* by applications (Dey, Salber, & Abowd, 1999). As mentioned earlier, context widgets differ from GUI widgets in that they live much longer, execute independently from individual applications, can be used by multiple applications simultaneously, and are responsible for maintaining a complete history of the context they acquire. Example context widgets include presence widgets that determine who is present in a particular location, Temperature widgets that determine the temperature for a location, sound level widgets that determine the sound level in a location, and activity widgets that determine what activity an individual is engaged in.

From a designer's perspective, context widgets provide abstractions that encapsulate acquisition and handling of a piece of context information. However, additional abstractions are necessary to handle context information effectively. These abstractions embody two notions: interpretation and aggregation.

Interpreters

Context interpreters are responsible for implementing the interpretation abstraction discussed in the requirements section (Section 3). *Interpretation* refers to the process of raising the level of abstraction of a piece of context. Location, for example, may be expressed at a low level of abstraction, such as geographical coordinates or at higher levels such as street names. Simple inference or derivation transforms geographical coordinates into street names using, for example, a geographic information database. Complex inference using multiple pieces of context also provides higher level information. For example, if a room contains several occupants and the sound level in the room is high, one can guess that a meeting is going on by combining these two pieces of context. Most often, context-aware applications require a higher level of abstraction than what sensors provide. *Interpreters* transform context information by raising its level of abstraction. An interpreter typically takes information from one or more context sources and produces a new piece of context information.

Interpretation of context has usually been performed by applications. By separating the interpretation out from applications, reuse of interpreters by multiple applications and widgets is supported. All interpreters have a common interface so other components can easily determine what interpretation capabilities an interpreter provides and will know how to communicate with any interpreter. This allows any application, widget, or aggregator to send context to an interpreter to be interpreted.

Aggregators

Aggregation refers to collecting multiple pieces of context information that are logically related into a common repository. The need for aggregation comes in part from the distributed nature of context information. Context must often be retrieved from distributed sensors, via widgets. Rather than have an application query each distributed widget in turn (introducing complexity and making the application more difficult to maintain), *aggregators* gather logically related information relevant for applications and make it available within a single software component. Our definition of context given earlier describes the need to collect related context information about the relevant entities (people, places, and objects) in the environment. Aggregators aid the architecture in supporting the delivery of specified context to an application, by collecting related context about an entity that the application is interested in.

Accessing information provided by aggregators is, therefore, a simplified operation for the application, and several applications can access the same information from a single aggregator. In addition, aggregators facilitate the inference of context information; inferring higher level of context from multiple, potentially distributed, pieces of context information is easier if all the pieces are available in an aggregator. By collecting logically related information, aggregators provide an abstraction that helps handle consistently separate pieces of context information, and offer a "one-stop shop" for applications.

For example, an application may have a context-aware behavior to execute when the following conditions are met: an individual is happy, located in his kitchen, and is making dinner. With no support for aggregation, an application (via the specification mechanism) has to use a combination of subscriptions and queries on different widgets to determine when these conditions are met. This is unnecessarily complex and is difficult to modify if changes are required. An aggregator is responsible for collecting all the context about a given entity. With aggregators, our application would only have to communicate with the single component responsible for the individual entity that it is interested in.

An aggregator has similar capabilities as a widget. Applications can be notified to changes in the aggregator's context, query/poll for updates, and access stored context about the entity the aggregator represents. Aggregators provide an additional separation of concerns between how context is acquired and how it is used.

Services

The three components we have discussed so far (widgets, interpreters, and aggregators) are responsible for acquiring context and delivering it to interested applications. If we examine the basic idea behind context-aware applications, that of acquiring context from the environment and then performing some action, we see that the step of taking an action is not yet represented in this conceptual framework. *Services* are components in the framework that execute actions on behalf of applications.

From our review of context-aware applications, we have identified three categories of context-aware behaviors or services (Section 2.3). The actual services within these categories are quite diverse and are often application-specific. However, for common context-aware services that multiple applications could make use of (e.g., turning on a light, delivering or displaying a message), support for that service within the architecture would remove the need for each application to implement the service. This calls for a service building block from which developers can design and implement services that can be made available to multiple applications.

A context service is an analog to the context widget. Whereas the context widget is responsible for retrieving state information about the environment from a sensor (i.e., input), the context service is responsible for controlling or changing state information in the environment using an actuator (i.e., output).

As with widgets, applications do not need to understand the details of how the service is being performed in order to use them.

Context services can be synchronous or asynchronous. An example of a synchronous context service is to send an e-mail to a user. An example of an asynchronous context service is to send a message to a user on a two-way pager containing a number of possible message responses. The service is complete when the user chooses a response and returns it. This is an asynchronous service because there is no guarantee when the user will respond and the application requesting the service probably does not want to wait indefinitely for a response. Instead, the service notifies the application that the service has been initiated and will deliver the service results when they are available.

Discoverers

Discoverers are the final component in the conceptual framework. They are responsible for maintaining a registry of what capabilities exist in the framework. This includes knowing what widgets, interpreters, aggregators, and services are currently available for use by applications. When any one of these components is started, it notifies a discoverer of its presence and capabilities and how to contact that component (e.g., language, protocol, machine hostname). Widgets indicate what kind(s) of context they can provide. Interpreters indicate what interpretations they can perform. Aggregators indicate what entity they represent and the type(s) of context they can provide about that entity. Services indicate what context-aware service they can provide and the type(s) of context and information required to execute that service. When any of these components fail, it is a discoverer's responsibility to determine that the component is no longer available for use.

Applications can use discoverers to find a particular component with a specific name or identity (i.e., white pages lookup). For example, an application may want to locate the Presence_Building_A Widget that contains information about the presence of people in Building A. Alternatively, applications can use discoverers to find a class of components that match a specific set of attributes and/or services (i.e., yellow pages lookup). For example, an application may want to access the aggregators for all the people that can be sensed in the local environment. Discoverers are similar to widgets in that they can notify applications when there are changes in the available set of components and can be queried or polled by applications.

Applications, and other context components, use the discoverers to locate context components that are of interest to them. Discoverers allow applications to not have to know a priori where components are located (in the network sense). They also allow applications to more easily adapt to changes in the context-sensing infrastructure, as new components appear and old components disappear.

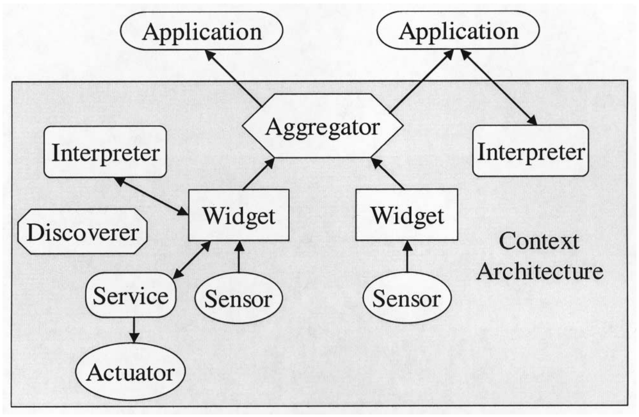
3.3. Using the Conceptual Framework

We have identified five categories of components that implement distinct functions. Context widgets acquire context information. Interpreters transform and raise the level of abstraction of context information. Interpreters may combine multiple pieces of context to produce higher level context information. Aggregators gather context information related to an entity for easy access by applications. Services execute behaviors on the environment using acquired context. Finally, discoverers allow applications (and other components) to determine the capabilities of the environment and to take advantage of them. A limited number of relations can be established between these components. Widgets are either queried or use notification to inform their clients of changes. Their clients can be applications, aggregators, or other widgets. Aggregators in turn act as a bridge between widgets and applications. Interpreters can be solicited at any stage and by any widget, aggregator, or application. Services are triggered primarily by applications (although other components may also use services). Discoverers communicate with all the components, acquiring information from widgets, interpreters, and aggregators, and provide this information to applications via notification or queries.

Figure 1 shows an example configuration including two sensing devices, two widgets, an aggregator, two interpreters, a service, a discoverer, and two applications. When any of the context components become available, they register their capabilities with a discoverer. This allows aggregators to find relevant widgets and interpreters and allows applications to find relevant aggregators, widgets, and interpreters. A sensor provides data to a context widget, which stores the context, can call an interpreter to obtain a higher level abstraction of the data, and then makes the context available to other components and applications. An aggregator collects context from widgets that can provide context about the entity it represents. It also stores the received context, can call an interpreter and then makes the context available to others. Finally, applications can query or subscribe to aggregators (or directly to widgets, if desired) and can call interpreters (if the desired level of abstraction is not available from the widgets and aggregators).

All of these components execute independently from applications, allowing for the constant acquisition of context and use by multiple applications. In addition, all components and applications communicate with each other automatically using known network protocols and languages. This allows a programmer who is implementing a particular component or an application to

Figure 1. **Example configuration of Context Toolkit components.**



communicate with other components without having to worry about the mechanisms used for communications.

We now demonstrate how the conceptual framework is used to build applications. In the next two subsections, we describe how the Active Badge call-forwarding application (Want et al., 1992), arguably the first context-aware application, and a mobile tour guide, the most common context-aware application, can be built.

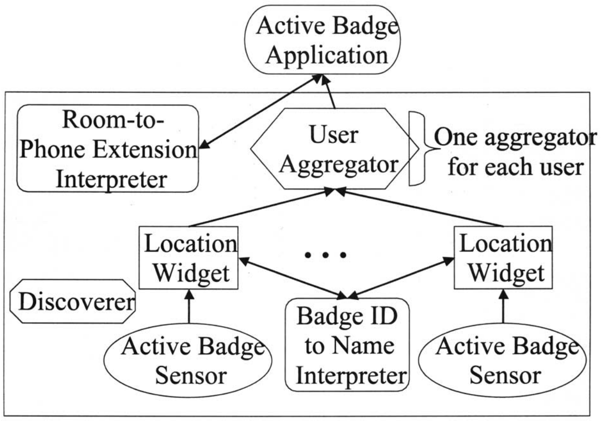
Active Badge Call-Forwarding

The original Active Badge application helps a receptionist to forward phone calls to the extension nearest the recipient's most recent location in a building. A dynamically updating table displays the phone extension and location for each person in the system. In addition, a user can:

- Request a list of the locations an individual has been in the past 5 min or past hour.
- Locate an individual and find the names of others in the same location.
- Request a list of the people at a particular location.
- Request to be notified when an individual is next sensed by the system.

Figure 2 shows how this application can be built with the conceptual framework described earlier. The Active Badge location system consisted of a number of infrared sensors distributed throughout a building that detected the presence of user-worn Active Badges. In the figure, a location widget represents each sensor. When a sensor detects a badge, its corresponding location

Figure 2. **Architecture diagram for the Active Badge call-forwarding application.**



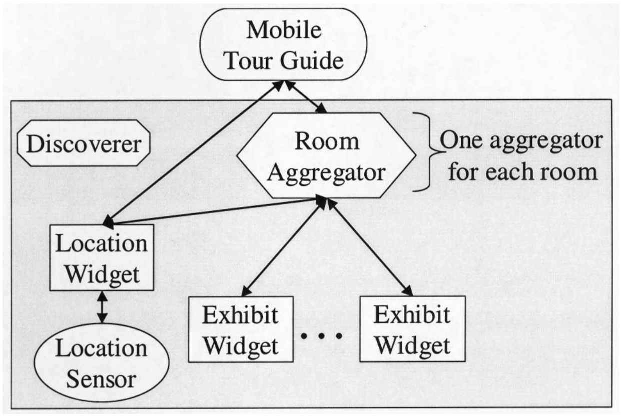
widget determines the wearer's identity (using the Badge ID to Name Interpreter), stores this information and passes it to the wearer's User Aggregator. Each User Aggregator (one for each user in the system) collects and stores all the location information from each of the location widgets for their corresponding user and makes it available to the application. All of these components (the interpreters, aggregators, and widgets) register with the Discoverer to allow them to locate each other as needed (widgets need to locate the Badge ID to Name Interpreter and the aggregator needs to locate the widgets). The application then uses the Discoverer to locate the User Aggregators and the Room-to-Phone Extension Interpreter.

To display a list of the building occupants, their location, and the nearest phone extension, the application just has to subscribe to all location updates in each User Aggregator and convert the room location to a local phone extension. Similarly, to be notified when an individual is next sensed by the system, the application can create a separate subscription to that individual's User Aggregator. For the other features (displaying all users in a particular location, locating an individual and others around him or her, and requesting a location history for a user), the application can poll the aggregators, specifying what information it is interested in.

Mobile Tour Guide

The most common context-aware application is the mobile tour guide (Abowd et al., 1997; Bederson, 1995; Davies et al., 1998; Feiner et al., 1997; Fels et al., 1998). A mobile tour guide typically runs on a handheld computer

Figure 3. **Architecture diagram for the mobile tour guide application.**



and allows a visitor to tour a particular indoor or outdoor space. As the visitor moves throughout the space, she is shown her current location on a map and is shown information about the interesting sites that she is near.

For the purposes of this design, we assume that we are building a mobile tour guide for an indoor space and that this space has a location system that reports location updates to a centralized server. Figure 3 shows the architecture diagram for this application. A location widget is built around the centralized location server. Each Room Aggregator (one for each room) subscribes to the location widget to be notified when a user enters and leaves its corresponding room. The Room Aggregators also subscribe to the exhibit widgets in their respective rooms so they know which of the exhibits are available at any given time. Each exhibit widget contains information about an exhibit on the tour. The application subscribes to the location widget for location updates for its user. When it receives an update, it updates its map display and polls the Room Aggregator representing the user's current location for a list of the available exhibits, if any. It displays the exhibit information to the user and also subscribes to the Room Aggregator to be notified of any changes to the exhibit list. If the application also displays the locations of the user's friends, it would create an additional subscription to the Room Aggregators for each friend to be notified when a location event corresponding to a friend is received.

4. THE CONTEXT TOOLKIT

In the previous section, we presented the requirements and component abstractions for a conceptual framework that supports the building and execution of context-aware applications. In this section, we discuss the details of the

Context Toolkit, our implementation of this conceptual framework. The Context Toolkit we have developed provides designers with the abstractions we have described—widgets, interpreters, aggregators, services, and discoverers—as well as a distributed infrastructure (Dey, Salber, et al., 1999; Salber et al., 1999). The Context Toolkit was developed in Java, although programming language-independent mechanisms were used, allowing the creation and interoperability of widgets, interpreters, aggregators, and applications in any language. We have instances of widgets and applications that have been written in C++, Frontier, Visual Basic, and Python.

Each widget, aggregator, interpreter, and discoverer component is implemented as a single process. Typically, different components are distributed on different processors. Thus, the Context Toolkit is built on top of a simple, distributed infrastructure that uses peer-to-peer communications. The main requirement of the distributed infrastructure is to support reliable communication between distributed objects across multiple platforms. Additional features such as multicasting, object migration, or resource discovery are desirable but not required.

4.1. Distributed Communications

In our current implementation of the Context Toolkit, we utilize a simple object communication mechanism based on HyperText Transfer Protocol (HTTP) and eXtensible Markup Language (XML) encoding of messages. This choice was dictated by a portability concern—the only requirement of our communication infrastructure is that the platform supports TCP/IP. With this simple requirement we can support the many unconventional or custom-built devices (such as wearable computers, two-way pagers, handheld computers, mobile phones, and off-the-shelf and custom sensors) that are common in ubiquitous or wearable computing environments. Possible alternatives such as Common Object Request Broker Architecture (CORBA) or Java Remote Method Invocation (RMI) can only be deployed on a limited number of platforms and require significantly more computing resources than our bare-bones approach. One drawback of our approach is its limited scalability when the number of components increases. However, we have not reached that limit yet in our experimentation, and solutions to this scalability problem exist in distributed infrastructures such as Jini™ from Sun® Microsystems (Sun Microsystems, 1999), at the expense of a more resource-intensive solution. The distributed communications ability is encapsulated in a component called BaseObject. Widgets and interpreters subclass from BaseObject and aggregators subclass from widgets (Figure 4), thereby inheriting this communications ability. The communications protocol and language were designed

Figure 4. **Object diagram for the Context Toolkit abstractions.**

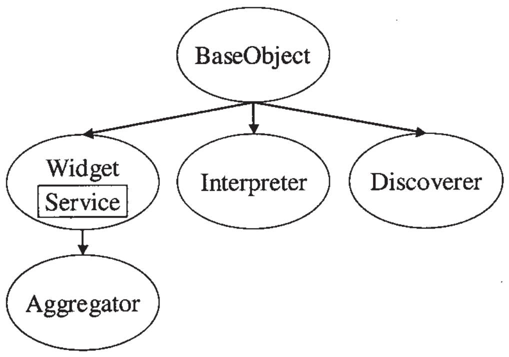
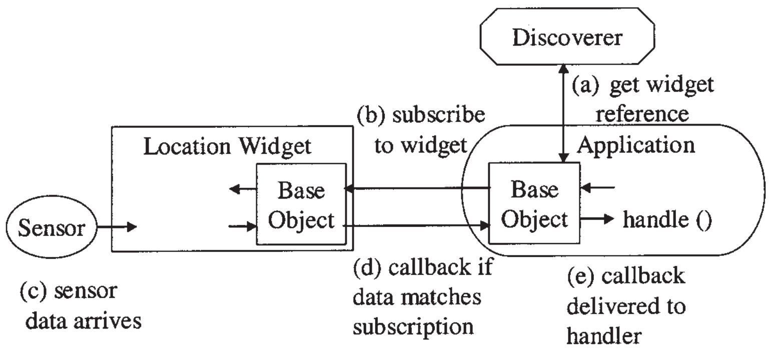


Figure 5. **Application subscribing to a context widget.**



to be pluggable, allowing instantiation-time replacement of both or either the protocol and language.

4.2. Subscriptions

Figure 5 demonstrates how an application subscribes to a context widget. First, an application needs to acquire a (programming-language independent) handle to the widget. It does so by (a) contacting the discoverer and specifying the context it is interested in (using either the white pages or yellow pages lookup mechanism). Once it has the widget handle, it (b) uses its instance of BaseObject to contact the widget. The widget's BaseObject receives the communication, strips off the HTTP and XML encoding, and passes the request to the widget. The request is to subscribe to the widget, so the widget adds the application to its list of subscribers, along with information about how to contact the application (the application's handle), the application method to call when context is received, and any filters on the subscription. An example of subscription filters are when an application does not want to receive all the context produced by a location widget but only wants context between 9 a.m. and 11 a.m. and only if it contains information about specified individuals. When the widget acquires new information from its sensor (c), it steps through its list of subscribers to determine to which subscriber(s) to send the context. If the acquired context matches the filters placed on the subscription for a particular subscriber, the context is delivered to the subscribing application. The context is sent from the widget using its BaseObject instance to the application's BaseObject (d). The application's BaseObject decodes the incoming message and calls the appropriate application method (e).

4.3. Event Handling

Although the concept of a context widget is modeled after GUI widgets, we emphasize an important difference in the way events are handled and distributed. When a piece of context information changes, the modification is detected by the context widget in charge of the specific piece of context. It timestamps the context and then notifies its subscribers, which may include other widgets, aggregators, and applications. Often the new information is modified by interpreters on the way from the context widget to the client subscriber. Because the context widget and its subscribers do not necessarily run on the same machine, events are sent through the distributed infrastructure as XML messages over HTTP. In a GUI windowing system, most events are generated by user actions and are expected to be sequential. They are placed in the event queue and processed sequentially. With distributed context, we cannot make any assumptions about event timings. Any component, context widgets, aggregators, and interpreters, as well as applications, must be ready to process events at any time, possibly simultaneously. The event queue mechanism clearly does not apply well to context. Rather than placing received events sequentially in a queue, all components listen for events constantly. When an event arrives, it is handed over to a new thread that processes the event (much like Web servers do). Thus, a component is always listening for events, and events are handled in parallel.

This approach of creating a new thread for each event is not always appropriate, particularly when designing for scalabilty. When widgets are delivering small amounts of context to applications, this approach works well. However, when streams of context are being sent (e.g., from a temperature sensor that produces updates each time the temperature changes by .01 degrees), this approach may be quite heavyweight. An alternative approach, which we have yet to implement, is to allow applications to specify an event handling policy to use, where creation of a new thread for each event would just be one of the available policies. The other obvious policy would be to create a single thread that would be responsible for handling streams of context from widgets.

An alternative model to the event delivery model chosen in the Context Toolkit would be to use a data flow model (Dey et al., 1998; Fox, 1998). In the data flow model, widgets act as sources of context and deliver context to applications that act as context sinks. Widgets also act as filters, by delivering only the context the application has indicated interest in. Interpreters and aggregators act as both context sinks that obtain context from widgets and context sources that provide context to applications and aggregators.

4.4. Discovery

The discovery mechanism used in the Context Toolkit is centralized. It uses only a single discoverer, as opposed to the federation of discoverers discussed in the conceptual framework. The centralized mechanism was chosen for reasons of simplicity, but suffers from a single point of failure. Like the communications mechanism, this mechanism was designed to be pluggable. It could easily be replaced with some more standard or sophisticated mechanism, such as the Service Location Protocol (SVRLOC Working Group of the IETF, 1999), Universal Plug and Play (Universal Plug and Play Forum, 2000), or Jini. When started, all widgets, aggregators, and interpreters register with the discoverer at a known network address and port. The discoverer "pings" each registered component at a prespecified frequency to ensure that each component is functioning correctly. If a component fails to respond to five consecutive pings, the component is removed from the registry and other components that have registered interest in this component are notified. In addition, when components are stopped manually, they notify the discoverer that they are no longer available to be used.

Applications do not register themselves with the discoverer. If an application is shut down manually, it will automatically unsubscribe from any widgets and aggregators it was subscribed to. However, if an application crashes, the subscribed-to widgets and aggregators will continue to attempt to send data to the application. If there is no response after five consecutive attempts, the application is removed from the subscription list.

4.5. Context Services

In the conceptual framework, context services are first class components. However, in our implementation of the framework, we chose to incorporate services into widgets (Figure 4). Now, rather than a widget being responsible for simply collecting data from a sensor, the widget also has the ability to perform the analog much like GUI interactors, to act on the environment using a transducer. We chose to combine these components, widgets, and services to present a logical functionality to programmers. For example, with a light widget, an application cannot only determine the current lighting conditions in a specified location, but can also change the lighting conditions (via lamps or window blinds). This allows a single component to logically represent "light" in that room.

The Context Toolkit we have built is an implementation of our conceptual framework for supporting the building and execution of context-aware applications. Although the particular choices we have made in our implementation have clear limitations with regards to scalability, our goal for supporting the rapid prototyping of applications outweighs the need for scalability at this time. The toolkit has enabled us to prototype and deploy a number of sophisticated context-aware applications, which we discuss in the next section. In addition, the toolkit has proven to be a research testbed that has allowed us to further explore difficult issues in the field of context-aware applications. We discuss these issues in Section 11.

4.6. Exploring the Space of Context-Aware Applications

We have identified relevant abstractions for dealing with context during design and corresponding components for managing context in context-aware applications. By providing a toolkit that handles most aspects of context acquisition, we facilitate the implementation of context-aware applications. However, the abstractions we provide assume designers can identify them easily from the application requirements. To help with this step, we now look at the actual use of the abstractions we have defined and suggest a design process for building context-aware applications.

We discuss five applications that we have implemented with the Context Toolkit, including the Conference Assistant described in Section 1.1. These applications have helped us to explore the design space of context-aware applications and to validate our conceptual framework and implementation. The applications will demonstrate that our context architecture is reusable, supports the acquisition and use of complex context beyond simple location and identity, and supports the evolution of context-aware applications.

5. IN/OUT BOARD AND CONTEXT-AWARE MAILING LIST: REUSE OF A SIMPLE WIDGET

We present two applications we have built that leverage off of the same context widget, one that indicates the arrival or departure of an individual from a physical space. We describe how each application is used and how each was built.

5.1. In/Out Board

Application Description

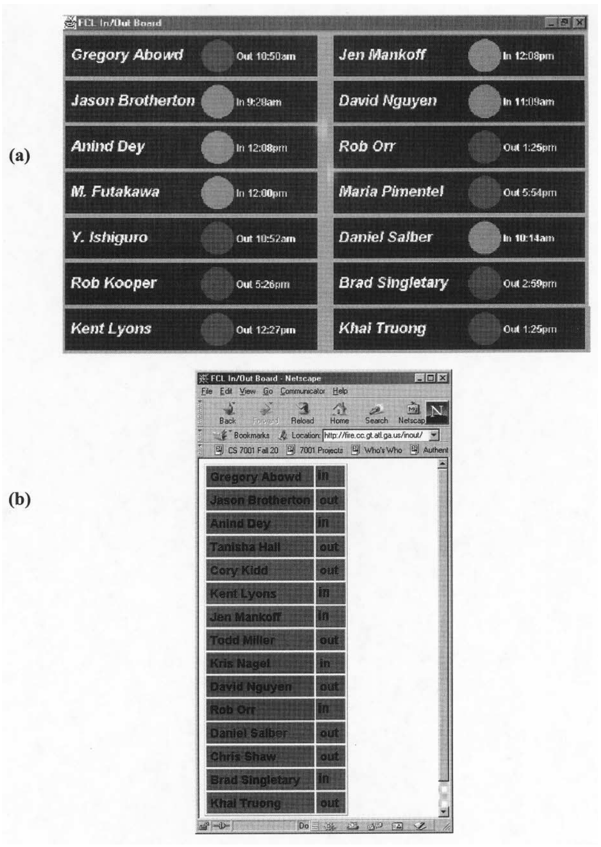
The first application we present is an In/Out Board. The In/Out Board built with the Context Toolkit is located on a laptop at the entrance to our research lab. It displays not only the in/out status (green/red dot) of building occupants, but also the day and time when the occupants last entered/left the building (Figure 6a). It is our longest running application, serving its users for over 2 years. A Web-based version of the In/Out Board displays the same information as the standard version with one exception. For privacy reasons, any requests for the Web version coming from a non–Georgia Tech IP address are only provided whether each occupant is in or out of the building, and not when they were last seen by the system (Figure 6b).

A third version of the In/Out Board application ran on a handheld Windows® CE device. This version used its own location to change the meaning of "in" and "out." For example, when a user is in Building A, anyone in the building is seen as "in" and those not in the building are seen as "out" on the interface. However, when the user leaves Building A and enters Building B, those in Building A are now marked as "out" and those who were in Building B, previously listed as "out," are now listed as being in the building. The device's location is tied to that of the user, so when the application knows the location of its user, it implicitly knows the location of itself.

Building the Application

Figure 7 shows the components that are used to build both the standard application and the Web-based application. The location widget detects users as they enter and leave our research area. We have two implementations of the location widget. One version (more commonly used, detects user presence when users explicitly dock their Dallas Semiconductor iButton™ (Dallas Semiconductor, 1999) into a reader. The widget uses this docking event to toggle the user's status, from "entered" to "left" or "left" to "entered." The second implementation uses the radio frequency-based PinPoint 3D-iD™ indoor RF-based positioning system (PinPoint, 1999) to automatically detect user presence. With this system, users wear pager-sized tags that are detected by antennas distributed about our research labs. When the tag is beyond the range of the antennas, the user's status is set to "left." Both technologies return an id (iButton or tag) that indicates which user was detected. An additional interpreter, the ID to Name Interpreter, is required to convert this id into a user's name.

Figure 6. **Standard (a) and Web-based versions of the In/Out Board application.**


FCL In/Out Board
(a)
(b)
FCL In/Out Board - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape
Bookmarks Location: http://fire.cc.gt.atl.ga.us/inout/
CS 7001 Fall 20 7001 Projects Who's Who Authent

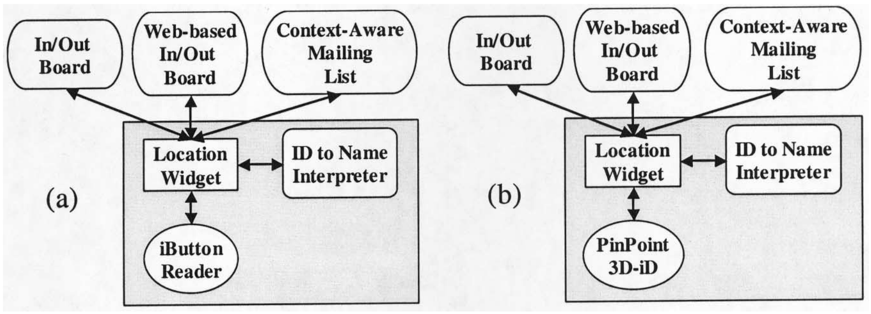
(a)

Gregory Abowd	Out 10:50am	Jen Mankoff	In 12:08pm
Jason Brotherton	In 9:28am	David Nguyen	In 11:09am
Anind Dey	In 12:08pm	Rob Orr	Out 1:25pm
M. Futakawa	In 12:00pm	Maria Pimentel	Out 5:54pm
Y. Ishiguro	Out 10:52am	Daniel Salber	In 10:14am
Rob Kooper	Out 5:26pm	Brad Singletary	Out 2:59pm
Kent Lyons	Out 12:27pm	Khai Truong	Out 1:25pm

(b)

Gregory Abowd	in
Jason Brotherton	out
Anind Dey	in
Tanisha Hall	out
Cory Kidd	out
Kent Lyons	in
Jen Mankoff	in
Todd Miller	out
Kris Nagel	in
David Nguyen	out
Rob Orr	in
Daniel Salber	out
Chris Shaw	out
Brad Singletary	in
Khai Truong	out

Figure 7. **Architecture diagrams for the In/Out Board and Context-Aware Mailing List applications, using (a) iButton readers and (b) PinPoint 3D-iD for location information.**



When the standard application is started, it queries the widget for the current status of each of the users listed on the In/Out Board. It uses this information to render the In/Out Board. It also subscribes to the widget for updates on future entering and leaving events. When a user arrives or leaves, the widget detects the event and notifies the subscribing application. Upon receiving the notification, the application updates the In/Out Board display accordingly.

The Web version is slightly different. It is a Common Gateway Interface script that maintains no state information. When the script is executed via a Web browser, it queries the widget for the current status of all users and automatically generates an HTML page containing the "in" and "out" status of users that the requesting browser can render. The requesting browser's IP address is used to control whether time information is included in the generated Web page.

5.2. Context-Aware Mailing List

Application Description

Using e-mail mailing lists is a useful way of broadcasting information to a group of people interested in a particular topic. However, mailing lists can also be a source of annoyance when an e-mail sent to the list is only relevant to a subgroup on the list. The problem is compounded when the subgroup's constituents are dynamic. The Context-Aware Mailing List is an application that delivers e-mail messages to members of a research group. It differs from standard mailing lists by delivering messages only to members of the research group who are currently in the building. This application supports the context-aware feature of automatically executing a service.

Building the Application

Figure 7 shows how the Context Toolkit was used to implement this application. The same widget and interpreter used in the In/Out Board application are used here. When the application is started, it queries the widget to find the current status of all of the users. The application adds each user who is currently in the building to a mailing list that is controlled by a standard majordomo mail program. The application then subscribes itself to the widget for updates on future entering and leaving events. When a user leaves the building, he is removed from the mailing list, and as a user enters the building, she is added to the mailing list. Now when a message is sent to the mailing list, it is only delivered to those members of the research group who are currently in the building.

5.3. Toolkit Support

The In/Out Board and the Context-Aware Mailing List are examples of simple applications that use simple context. Both applications required the same type of context, presence of an individual in defined location, allowing reuse of the context widget that senses presence. Reuse of context components eases the development of context-aware applications. We have built a third prototype application that reuses these same components, an Information Display. It simply displays information thought to be relevant to the user when she arrives in the building. Relevance was decided based on the research group that the user was a part of. Even with just this set of simple components, a number of interesting context-aware applications can be easily built.

The context component abstraction also made it very easy for us to change the underlying technology used to sense presence information. We were able to swap the widget implementation entirely when we went from using iButton technology to using PinPoint technology and *vice-versa*, without changing a line of code in either application. In addition, both technologies could be used simultaneously by both or either application. These abilities allow us to easily evolve our context-aware systems, in terms of the sensing technologies, and to prototype with a variety of sensors.

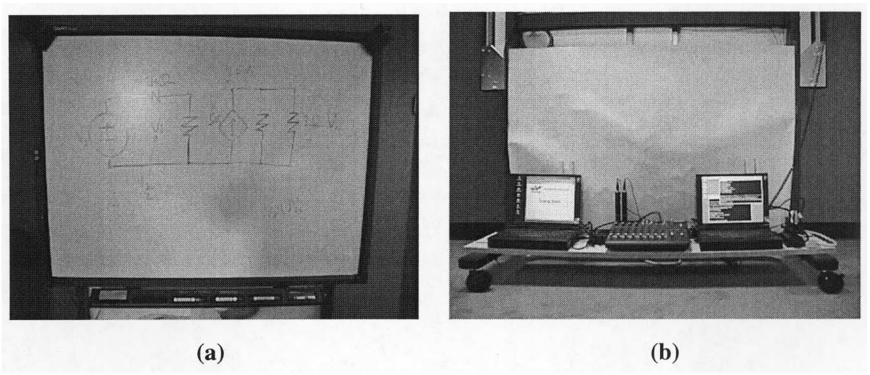
The applications used context dealing with users' presence within a research laboratory. The context-sensing environment in our lab only contained a single widget, at any time, that could provide this information. Ideally, multiple context widgets using homogenous or heterogeneous sensing technologies (mouse/keyboard activity, office door opening, intelligent floor, etc.) would be available to sense presence information within the lab. If there were, the applications would have obtained the required context from an aggregator that represented the research lab. The applications' code would not look very different, communicating with an aggregator instead of a widget, but the applications would be enhanced by being able to leverage off of multiple widgets and by being more insulated from run-time widget instantiations and crashes.

6. DUMMBO: EVOLUTION OF NON-CONTEXT-AWARE APPLICATIONS

6.1. Application Description

Dynamic Ubiquitous Mobile Meeting BOard (DUMMBO) was an already existing system (Brotherton et al., 1999) that we chose to augment. DUMMBO is an instrumented digitizing whiteboard that supports the capture and access of informal and spontaneous meetings (Figure 8). Captured meetings consist of the ink written to and erased from the whiteboard as well as the recorded au-

Figure 8. **DUMMBO: Dynamic Ubiquitous Mobile Meeting BOard. (a) Front view of DUMMBO. (b) Rear view of DUMMBO. The computational power of the whiteboard is hidden under the board behind a curtain.**



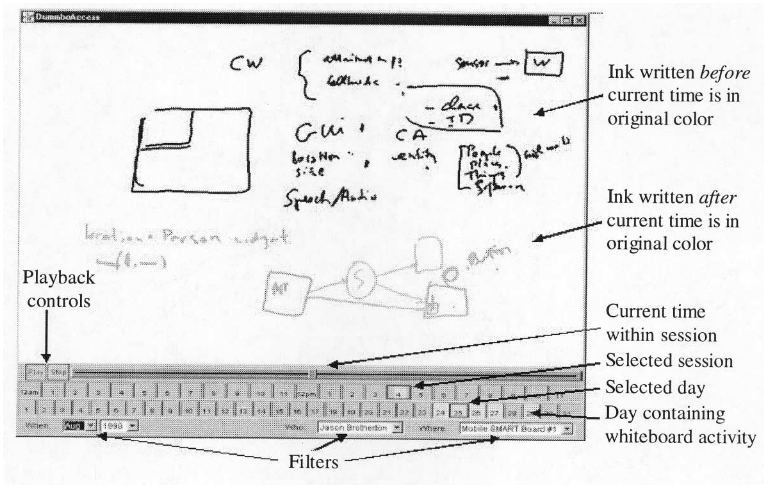
dio discussion. After the meeting, a participant can access the captured notes and audio by indicating the time and date of the meeting.

In the initial version of DUMMBO, the recording of a meeting was initiated by writing strokes or erasing previously drawn strokes on the physical whiteboard. However, by starting the audio capture only when writing began on the whiteboard, all discussion that occurred before the writing started was lost. To enable this discussion to be captured, we augmented DUMMBO to have its audio recording triggered when there are people gathered around the whiteboard.

By keeping track of who is around the whiteboard during capture and the location of the whiteboard, we are able to support more sophisticated access of captured material. Rather than just using time and date to locate and access previously captured meetings, users can also use the identities of meeting participants, the number of participants, and the location of the meeting to aid access.

Figure 9 is a screenshot of the DUMMBO access interface. The user can specify information such as the approximate time and location of the meeting. A timeline for the month is displayed with days highlighted if there was whiteboard activity in that day at that place. The interface also indicates the people who were present at the board at any time. Above the month timeline is an hour timeline that shows the actual highlighted times of activity for the selected day. Selecting a session then brings up what the whiteboard looked like at the start of the session. The user can scroll forward and backward in time and watch the board update. This allows for quick searching for a particular time in the meeting. Finally, the user can access the audio directly from some writing on the board, or request synchronized playback of ink and audio from

Figure 9. **DUMMBO access interface. The user selects filter values corresponding to when, who, and where. DUMMBO then displays all days containing whiteboard activity. Selecting a day will highlight all the sessions recording in that day. Playback controls allow for live playback of the meeting.**

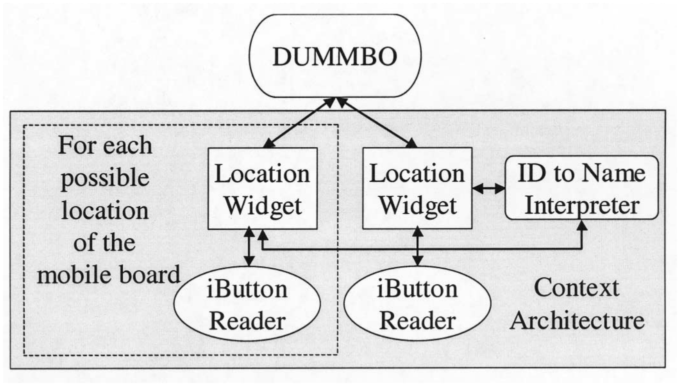

Ink written *before* current time is in original color
Ink written *after* current time is in original color
Playback controls
Current time within session
Selected session
Selected day
Day containing whiteboard activity
Filters

any point in the timeline. The interface supports a more efficient and intuitive method for locating and browsing captured meetings than was previously available. This application supports the context-aware features of automatically executing a service (starting audio recording) and of tagging of context to information for later retrieval.

6.2. Building the Application

The context architecture used to support the DUMMBO application is shown in Figure 10. It used a single location widget to detect which users were around the mobile whiteboard. It used multiple location widgets to determine where the whiteboard was located in our lab—there was one widget for each location where DUMMBO could be moved. When the application is started, it subscribes to all of the location widgets. For the single widget that the application uses to detect users around the whiteboard, the application subscribes for all available events (anyone arriving or leaving). However, for the multiple widgets that the application uses to determine the location of the whiteboard, the application subscribes to filtered events, only being interested in the whiteboard arriving or leaving from a particular location. When the application receives notification from the widget that there are people around the whiteboard, it starts recording information. When it receives notification

Figure 10. **Context architecture for the DUMMBO application.**



about a change in the whiteboard position, it augments the meeting location tag associated with the meeting being captured.

6.3. Toolkit Support

As mentioned previously, DUMMBO was an existing application that was later augmented with the use of context. DUMMBO was augmented by the original researchers who built it (other members of our research group), and not by researchers involved with the Context Toolkit. These researchers simply needed to install the Context Toolkit, determine which widgets they were interested in, instantiate those widgets, and handle the widget callbacks. In all, the application only required changing/adding 25 lines of Java code (out of a total of 892 lines for the main application class file) and modifications were localized in a single class file. The significant modifications include 4 lines added to use the Context Toolkit and widgets, 1 line modified to enable the class to handle widget callbacks, and 17 lines that provided the desired context-aware behavior. Comparatively, the size of the Context Toolkit, at that time, was about 12,400 lines of Java code. By supporting the use of context without programmers having to add large amounts of code, the Context Toolkit has made it easier to leverage from context.

Unlike the In/Out Board and the Context-Aware Mailing List, this application used multiple widgets. All of the widgets were of the same type, providing presence information. However, all of the widgets did not have to use the same underlying sensing technology. Each widget could have used a different sensor and the application would not require any changes.

In this application, using an aggregator would have reduced the number of lines of code required by the programmer. An aggregator that represented the mobile whiteboard could collect all the location information from each of the presence widgets, allowing the application to only subscribe to the aggregator for knowledge of the whiteboard's location. The tradeoff with reducing the number of lines of application code is in creating an aggregator for the whiteboard if one did not already exist and executing an additional context component.

7. INTERCOM: USE OF COMPLEX CONTEXT AND SERVICES

7.1. Application Description

Intercoms in homes are intended to facilitate conversations between occupants distributed throughout the home. Standard intercoms have drawbacks including the need to move to a particular location in the home (e.g., a wall-mounted intercom unit) or carry a handset to use the intercom, the lack of control in directing communication to an occupant without knowing his or her location (e.g., broadcasting is often used) and the lack of knowledge of the callee's status to determine whether the communication should occur. The Intercom application uses context acquired from an instrumented home to facilitate one-way (i.e., monitor or broadcast) and two-way (i.e., conversations) communications to address these drawbacks (Kidd, O'Connell, Nagel, Patil, & Abowd, 2001).

An occupant of the home can use the Intercom application simply by talking to the home. For example, to broadcast a message to all other occupants, the initiator can say, "House, I would like to announce… ." The house responds with "Go ahead" to indicate that it has understood the request, and the user continues with "Dinner is ready so everyone should come to the kitchen." This audio announcement is delivered to every occupied room in the house. To indicate the announcement is over, the user says, "Stop the intercom."

If a parent wants to use the intercom to facilitate baby monitoring, she can say, "House, how is the baby doing?" The intercom delivers the audio from the room the baby is in to the room that the mother is in. As the mother moves throughout the house, the audio from the baby's room follows her. She can stop monitoring by saying, "Stop the intercom."

Finally, if an occupant of the house wants to initiate a conversation with another person in the house, he can say, "House, I want to talk to Sam." The house responds by telling the occupant that Sam is currently in the living room with Barbara and asks whether the occupant still wants to speak with Sam. The user really needs to talk to Sam and the details of the conversation will not be personal, so he approves the connection. If there were no one in the room with Sam, the approval would be automatic. The Intercom application sets up a two-way audio routing between the occupant's room and the living room. If either the conversation initiator or Sam change rooms during the conversation, the audio is correctly re-routed to the appropriate rooms.

7.2. Building the Application

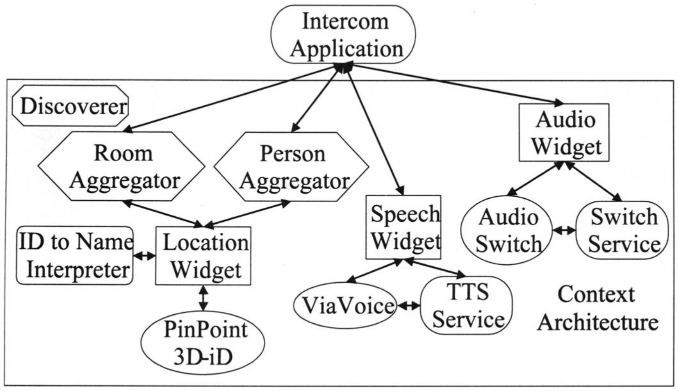
The context architecture used to support the Intercom application is shown in Figure 11. A speech widget wrapped around IBM® ViaVoice™ (IBM, 2000) was used to recognize what users were saying when speaking to the house and to support a Text-To-Speech (TTS) Service to allow the house to speak to the users. Ceiling-mounted speakers were placed in each room to provide audio output and users wore wireless microphones to provide the input to the house (and ViaVoice). An audio widget was built around an audio switch, supporting a Switch Service that enabled the routing of audio throughout the house. A location widget that used the PinPoint 3D-iD positioning system was used to determine where users were in the house. Room Aggregators (one for each room in the house) and Person Aggregators (one for each occupant) collect location information from the location widget.

The Intercom application subscribes to the speech widget to be notified of any requests (containing the type of connection, caller, and callee) to use the intercom. Upon receiving a request, the application queries the Person Aggregator for the caller and callee to determine their respective locations and, in the case of a conversation request, queries the Room Aggregator for the callee to determine if anyone else is in the room. The application uses the TTS Service and the Switch Service to provide the caller with information (e.g., ready to hear the announcement or information about other people with the callee). If the connection is approved by the caller (when necessary), the Switch Service is used again to route the audio appropriately, depending on the type of connection. Then, the application subscribes to the relevant Person Aggregators to be notified of either the caller or callee changing rooms. If either changes rooms, the Switch Service is used to re-route the audio. If one party enters a room where there is already an Intercom-facilitated conversation occurring, the Switch Service is not executed until the party enters an unused room. Finally, the application uses the Switch Service to end the connection when notified by the speech widget that one of the conversation participants has terminated the call, or when notified by the Person Aggregators that the participants are in the same room.

7.3. Toolkit Support

The Intercom application is a fairly sophisticated context-aware application that was built by other members of our research group that did not include

Figure 11. **Architecture diagram for the Intercom application.**



any developers of the Context Toolkit. It displays context to the user when indicating the presence of another individual in the callee's room and automatically executes services using context when creating audio connections between users and re-routing them as users move throughout the house. This application uses context beyond the simple location of an individual. It uses information about what users' say, co-location of users and the status of other conversations in the house. The Intercom application also demonstrates the use of aggregators to simplify application development and the use of services to allow actuation in the environment.

The developers of this application were able to reuse the location widget, speech widget, and the Room and Person Aggregators from previously developed applications, leaving only the audio widget and the application to be developed. Reuse of these components made the design and implementation of the Intercom system much easier.

8. CONFERENCE ASSISTANT: USE OF COMPLEX CONTEXT AND SYSTEMATIC DESIGN OF A CONTEXT-AWARE APPLICATION

In this section, we describe in detail the use of the abstractions we have introduced, as well as a design methodology with an actual application that we have built. The context-aware application we consider is a Conference Assistant. Its use is illustrated in the scenario of Section 1.1. As implemented, it is not quite as rich as the scenario, but the differences are not because of any deficiency in the Context Toolkit or the conceptual framework. We have implemented a scaled-down prototype as a proof of concept and to test our ideas

(Dey, Futakawa, Salber, & Abowd, 1999). To simulate our conference setting, we defined three presentation areas in our laboratory area, had a number of colleagues act as presenters in the three areas, and had six other colleagues act as conference attendees. We first present a methodology for identifying context components from application requirements, provide details on the typical use of the Conference Assistant, and show how to apply our methodology on this particular example. The end result of the process is the software architecture of the application.

8.1. Identifying Context Components

The context components that we presented map nicely to concepts we introduced in Section 2. In Section 2.1, we introduced entity categories such as places, people, and objects, and in Section 2.2, we introduced categories of context information. In the design of a context-aware application, designers identify relevant entities and, using the categories of context information, identify for each entity relevant context attributes. In addition, designers must specify requirements for all pieces of context information. These requirements typically express either the desired frame of reference (e.g., units or naming scheme) or quality of service constraints (e.g., desired coverage or resolution for location). Then, the following mapping rules provide a straightforward way to identify the context components of the architecture:

1. For each context attribute, a context widget acquires the required context information.
2. For each context entity, an aggregator collects context information about the entity from context widgets.
3. Applications obtain necessary context from relevant context entities.
4. When pieces of context information must be used to provide higher level context information, interpreters assure the inference.
5. When discrepancies occur between the requirements of context attributes and the information provided by sensors with regard to frames of reference, interpreters assure the conversion from one frame of reference to another.
6. When applications receive the context they are interested in, they can either perform application-specific behaviors or use available services in the environment.

When the available components in the environment do not meet the needs of the application being built, it is up to the application programmer to implement the missing components. The advantages of building the components using the abstractions rather than adding functionality to the application in an ad hoc manner are:

- *Ease of building* by leveraging off of existing components. For example, to build a widget, only the details specific to the instance being created need to be provided by the programmer, including descriptions of the context the widget is acquiring and how to communicate with the sensor being used. Storage of context, support for notification and querying, and communications with other components are automatically provided.
- *Support for reuse.* By implementing the functionality outside the application, multiple applications can reuse the new components, either one at a time or simultaneously, as shown in Section 5.
- *Support for evolution.* Context-aware application evolution occurs in three ways, either by changing the context the application uses, the way the context is acquired, or the behaviors that are executed when the application receives the appropriate context. By using the described component abstractions, applications can change the context they use by changing the specifications they provide to a discoverer that will provide them a handle to an appropriate component (Section 6). If a widget changes the sensing technology it uses to acquire context, applications that use the widget are not affected and do not need to be changed (Section 5). Finally, if the application changes the behaviors it wants performed, it can simply change which service component it uses.

8.2. The Conference Setting

Our desire to build context-aware applications that deal with a wide variety of context (moving beyond simple identity and location) led us to investigate a complex setting with rich sources of context. We chose the environment of a conference, which involves several people, multiple locations, and a diverse collection of activities. We performed an informal study of what conference attendees do at conferences and came up with the following list:

- Determine what presentations/demonstrations are interesting/relevant and attend those.
- Keep track of colleagues to share relevant information.
- Take notes on interesting presentations.
- Meet other people doing interesting research.

The Conference Assistant is a context-aware application that supports conference attendees in performing the first three tasks—the last task will be supported in a subsequent version of the application. Because the conference setting is such a dynamic environment, context awareness has much to offer. As described in the next section, context awareness allows the Conference Assistant to adapt to the changing context, providing additional and value-added support for the aforementioned tasks.

8.3. Using the Conference Assistant

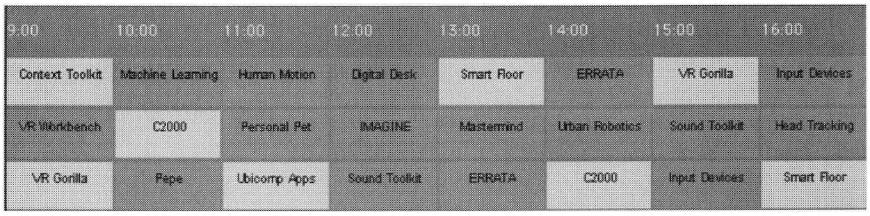
The user is attending a multitrack conference. When she registers, she receives a handheld device that runs the Conference Assistant application. When she starts the application, it automatically displays a copy of the conference schedule, showing the multiple (three) tracks of the conference, including both paper tracks and demonstration tracks. The schedule is augmented, highlighting certain papers and demonstrations that are highlighted (light gray) to indicate they may be of particular interest to the user (Figure 12). The Conference Assistant compares the user's research interests, provided during registration, against the text in the abstract of each presentation to find potential matches.

The user takes the advice of the application and walks toward the room of a suggested paper presentation. When she enters the room, the Conference Assistant automatically displays the name of the presenter and the title of the presentation. It also indicates whether audio and/or video are being recorded for this presentation. The presenter is using a combination of PowerPoint^{TM} and Web pages. A thumbnail of the current slide or Web page is displayed on the handheld computer display. The Conference Assistant allows the user to create notes of her own to attach to the current slide or Web page (Figure 13). As the presentation proceeds, the application displays updated slide or Web page information.

The user looks at the conference schedule and notices that her colleagues are attending other presentations (Figure 14). A colleague (Daniel) has indicated a high level of interest in a particular presentation (Digital Desk), so she decides to join him. This display of colleagues' context supports a type of dynamic social and collaborative filtering.

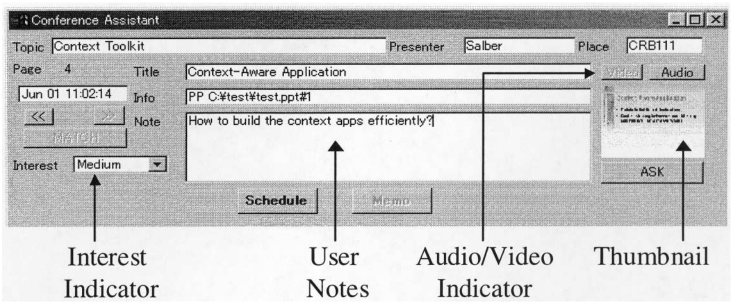
After the conference, she uses a retrieval application, running on her desktop computer, to retrieve information about a particular presentation. The application shows her a timeline of the conference schedule with the presentation and demonstration tracks (Figure 15a). The application uses context-based retrieval. It provides a query interface that allows the user to populate the timeline with various events—her arrival and departure from different rooms, when she asked a question, when other people asked questions or were present, when a presentation used a particular keyword, or when audio or video were recorded. By selecting an event on the timeline (Figure 15a), the user can view (Figure 15b) the slide or Web page presented at the time of the event, any text that could be captured from the slide or Web page, any audio

Figure 12. **Screenshot of the augmented schedule with three (horizontal) tracks. This screenshot comes from a handheld Windows CE device with a resolution of 640 × 240 pixels.**



9:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00
Context Toolkit	Machine Learning	Human Motion	Digital Desk	Smart Floor	ERRATA	VR Gorilla	Input Devices
VR Workbench	C2000	Personal Pet	IMAGINE	Mastermind	Urban Robotics	Sound Toolkit	Head Tracking
VR Gorilla	Pepe	Ubicomp Apps	Sound Toolkit	ERRATA	C2000	Input Devices	Smart Floor

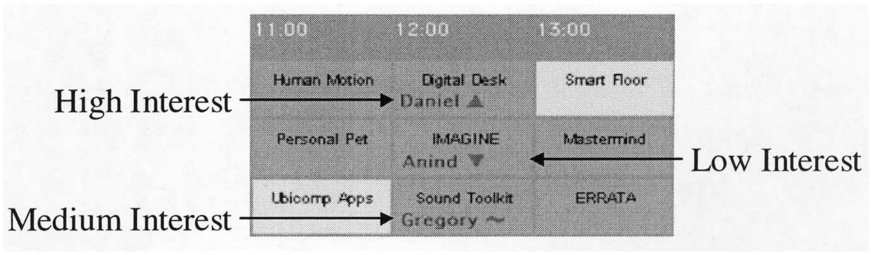
Figure 13. **Screenshot of the Conference Assistant note-taking interface.**



Conference Assistant

Topic: Context Toolkit
Presenter: Salber
Place: CRB111
Page: 4
Jun 01 11:02:14
Title: Context-Aware Application
Info: PP C:¥test¥test.ppt#1
Note: How to build the context apps efficiently?
Interest: Medium

Figure 14. **Screenshot of the partial schedule showing the location and interest level of colleagues. Symbols indicate interest level.**

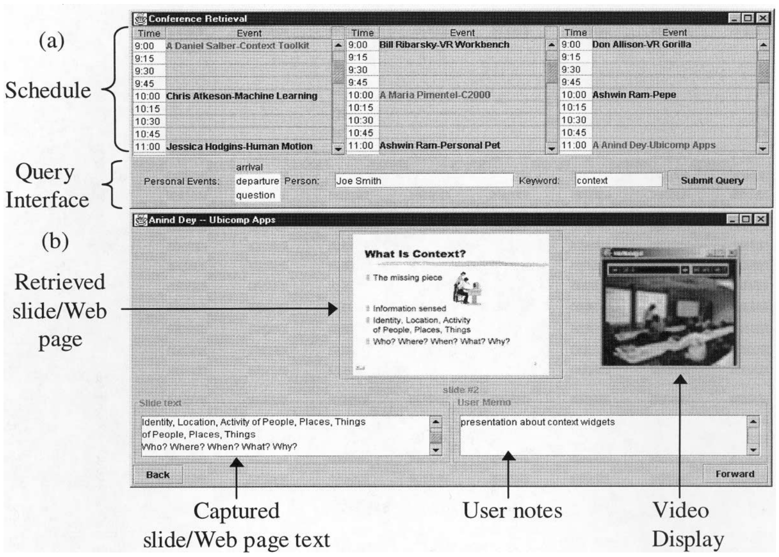


11:00	12:00	13:00
Human Motion	Digital Desk Daniel ▲	Smart Floor
Personal Pet	IMAGINE Anind ▼	Mastermind
Ubicomp Apps	Sound Toolkit Gregory ~	ERRATA

and/or video recorded during the presentation of the slide, and any personal notes she may have taken. She can then continue to view the current presentation, moving back and forth between the presented slides and Web pages.

In a similar fashion after the conference, a conference presenter can use this application to retrieve information about her presentation. The application

Figure 15. **Screenshots of the retrieval application: query interface and timeline annotated with events (a) and captured slideshow and recorded audio/video (b).**



Time	Event	Time	Event	Time	Event
9:00	A Daniel Salber-Context Toolkit	9:00	Bill Ribarsky-VR Workbench	9:00	Don Allison-VR Gorilla
9:15		9:15		9:15	
9:30		9:30		9:30	
9:45		9:45		9:45	
10:00	Chris Atkeson-Machine Learning	10:00	A Maria Pimentel-C2000	10:00	Ashwin Ram-Pepe
10:15		10:15		10:15	
10:30		10:30		10:30	
10:45		10:45		10:45	
11:00	Jessica Hodgins-Human Motion	11:00	Ashwin Ram-Personal Pet	11:00	A Anind Dey-Ubicomp Apps

Personal Events: arrival / departure / question Person: Joe Smith Keyword: context

What Is Context?

- The missing piece
- Information sensed
- Identity, Location, Activity of People, Places, Things
- Who? Where? When? What? Why?

slide #2

Slide text:
Identity, Location, Activity of People, Places, Things
of People, Places, Things
Who? Where? When? What? Why?

User Memo:
presentation about context widgets

again displays a timeline of the presentation, differing only with the events that populate the timeline. In this case, the events include when different slides were presented, when audience members arrived and left the presentation (and their identities), the identities of audience members who asked questions and the slides relevant to the questions. The presenter can “re-live” the presentation by playing back the audio and/or video and moving between presentation slides and Web pages.

8.4. Applying the Design Methodology

Starting from the usage scenarios presented earlier (Section 1.1), we must first identify the entities that are likely to provide context information to the Conference Assistant application. We then look at their context attributes, and the requirements for each attribute. Then we examine our choice of sensors to acquire the specified context information. Finally, we derive a software architecture for the application from this information on context sources and attributes.

Identifying Entities

Object-oriented analysis (OOA) methods provide useful support to identify objects. Among these objects, users are usually context entities. In the case of the Conference Assistant, both attendees and presenters are of interest. Places, rooms for the Conference Assistant, are also modeled as context entities. Physical and computational objects must be handled with care; good candidates for context entities are objects that are highly dynamic. The conference schedule for example, or a conference track, are static objects and do not qualify. Indeed in our scenario, we do not account for changes in the conference schedule, although that would be an obvious extension. On the other hand, a presentation, or the current slide or Web page displayed by the presenter, are objects of interest for context modeling. Also of interest are objects whose content we wish to capture and annotate with context information, such as a question asked by an attendee, or notes taken by the user.

Identifying Context Attributes

Once we have identified context entities, we look at candidate context attributes for each entity. Again OOA methods can provide insight by identifying attributes of context entities' corresponding objects. Then, we must decide whether a given attribute is context related. To identify context-related attributes, we use the context categories of Section 2.2 as a guide. For each entity, attributes that fit into context categories are interesting context attributes. For example, for an attendee, contact information, interests, and colleagues are derived from identity. The room where the attendee currently stands is location information. The time when the user moves into and out of room is needed by the retrieval application that shows the user a timeline of her whereabouts. Finally the current level of interest of the user falls into the status category. For a presenter, we only need the name of the person (i.e., her identity).

When it comes to the room entity, we notice something interesting. The room is only used to characterize the location of the user, and the location where a presentation is held. As a matter of fact, our scenario does not involve attributes of the room per se, except its identity. Because we have to distinguish between the different rooms where presentations are being held, we need to identify the rooms in some way. However, one could imagine that a different application would involve context attributes of the room. For example, we might be interested in monitoring the headcount in all rooms at all times, to help planners for the next edition of the conference. In that case, we would identify an "occupancy" attribute for each room.

For a presentation, we are concerned with its location (i.e., the room it is held in). The current slide or Web page of a presentation has an identity (its slide number or URL), secondary identity information (a title), and status information (a thumbnail shown on the user's handheld, according to the scenario).

Question and user notes share the same attributes—both the slide or Web page they relate to and the time at which the question was asked or the note

Figure 16. **Context entities and attributes for the Conference Assistant application.**

Entities	Attributes
People	
Attendee	Name (contact info, interests, colleagues) Room and times of arrival/departure Level of interest
Presenter	Name
Places	
Room	Name
Things	
Presentation	Room, time
Slide	Number, title, thumbnail, time
Web page	URL, title, thumbnail, time
Question	Related slide or Web page, time
User notes	Related slide or Web page, time

Note. Attributes are derived context types.

was taken are of interest. Question or note contents are captured and annotated with context information. We need to be able to relate the note to either a slide or Web page and display the note in the timeline provided by the retrieval application. Similarly, we annotate questions with time information and the slide or Web page the question was about (if any).

Figure 16 summarizes the identification of context entities and their context attributes for the Conference Assistant application.

Identifying Quality of Service Requirements

We now look at the quality of service constraints for each context attribute. In this part of the analysis, we focus on dynamic attributes. The names of the attendee and presenter as well as the room name or the room a presentation occurs in do not present particular challenges with regard to quality of service. They are set once and for all, and they might be acquired once by the application at startup. On the other hand, the location of the user is highly dynamic, as well as the context attributes of the current slide or Web page, and the level of interest of the user.

The user's location must be available to the application whenever the user is on the conference grounds. Thus, we require that coverage for the user's location is at least the space occupied by the conference. This space does not need to be contiguous. In our demonstration setup in our lab, we used two areas at the opposite ends of the lab and a room across the corridor as our three rooms where the three tracks of a simulated conference are held. The requirements are less strict for resolution: We want to be able to differentiate the user's location among the three rooms she's visiting. But the location of the user within the room is not useful in our scenario. Thus, room-level resolution is sufficient. However, if we extended the application to help conference attendees navigate between different rooms, a much higher level of resolution would be required. We must specify high accuracy and reliability requirements: Incorrect or missing user location would seriously jeopardize the usefulness of the Conference Assistant. For frequency and timeliness, our requirements must match the expected behavior of the users. We expect attendees to move at a normal pace between rooms and stay in a room several minutes at least. The retrieval application, which will allow users to re-live their experience, displays a schedule whose resolution is no more than 5 min. But when a user moves into a room, we want to display almost immediately the information about the presentation currently held in the room. The user might accept a delay of a couple of seconds for this information to be displayed on her handheld. As a consequence, we may set our requirements for frequency at one update every second minimum and timeliness at 1 sec maximum.

Context attributes of a slide or a Web page are presented to the user on the handheld device. They are attached to notes taken by the user. Coverage and resolution apply to the slides' context attributes. Partial coverage would mean the context information available to the application would be constrained (e.g., only slides with no color pictures, or titles shorter than 10 characters). Limited resolution would mean, for example, that only every other Web page jump would be transmitted to the application. Of course, either limited coverage or limited resolution would cripple the Conference Assistant. We thus require 100% coverage and the best possible resolution. We also set high requirements for accuracy and reliability for similar reasons. Finally, we must evaluate our needs in terms of update frequency and timeliness. As a first approximation, we can set both the update frequency and timeliness to about 1 sec. We want to avoid glaring discrepancies between the display shown to the audience and the display presented on the handheld. When the application is being deployed, these quality of service requirements would most definitely need to be backed with theoretical data or user testing.

The level of interest is set by the user and is chosen among three values (low, medium, and high). The level of interest is communicated to colleagues of the user. Because the level of interest is a discrete variable with very few values, required coverage and resolution are 100% and best available, respectively. Similarly, very high accuracy is required. However, we can tolerate less than perfect reliability. This feature is not essential for the operation of the Conference Assistant. Frequency and timeliness are also not of the highest importance, and a typical figure of a few minutes for both would be acceptable.

Choosing Sensors

At this stage of the analysis, we have identified the pieces of context that we need to acquire and the quality of service requirements for each of them. We now examine the sensing techniques available for each context attribute and use quality of service requirements to choose an appropriate set of sensing techniques.

Except for the user's location, our choice of sensing techniques is severely limited. Information about the current slide or Web page can only be acquired through interapplication communication mechanisms such as COM/DCOM. The level of interest of the user is input manually by the user. Other techniques could be used but were just not practical in our setup. Computer vision techniques could extract relevant information from a slide, although reliability may be an issue. Physiological sensors may be able to determine the level of interest of the use, although reliability may also be an issue, as well as the intrusiveness of the sensors.

We have a number of techniques available for acquiring the user's location. Among them, we considered Olivetti's Active Badges (Want et al., 1992), the Dallas Semiconductor iButtons, and PinPoint 3D-iD. All systems satisfy our requirements in terms of coverage, resolution, and accuracy. With Active Badges, users wear IR-transmitting badges whose signal is picked by fixed receivers. With iButtons, users carry iButtons devices and dock them explicitly to indicate their change of location; adding docks extends the coverage and a room-level resolution is easily achieved. With the PinPoint system, users carry a RF active tag whose signal is received by a set of antennas (seven in our setup). Triangulation between the signals received at the antennas provides the tag's location. We have set high requirements for reliability, but all systems have potential drawbacks. Active Badges require line of sight between the badge and a reader installed indoors. In a room potentially packed with attendees, we have no guarantee that the transmission will be successful. In this regard, a RF-based system like the PinPoint system is preferable. But it suffers from other potential reliability problems: The tag orientation is crucial for its operation. Providing a carefully designed tag holder could alleviate this problem. Finally, the iButtons suffer from another kind of reliability problem: Because they are operated manually by the user, we run the risk that users will forget to indicate their change of location. In our experience with iButtons and the PinPoint system, they both satisfy our frequency and timeliness requirements. In summary of this analysis, the PinPoint system is the most appropriate given our requirements. However, the complexity involved in the initial setup of the PinPoint system led us to use iButtons in our first prototype. The iButtons provide one significant advantage over the PinPoint system, that of providing the users control over disclosing their location to the system and others. Only motivated users, those that want to collaborate with their colleagues or want to take advantage of the advanced services that require user location, will provide their location. Both systems provide slightly different information. The PinPoint system provides the identity of the tag as a numeric id and a zone number, according to a zone map defined statically. With the iButtons, each dock provides the identity of the docking iButton as a numeric id. When using several docks, we attach location information to each dock.

Deriving a Software Design

To derive a software design from the analysis of context entities and attributes, we apply the rules introduced in Section 3.3. We first map entities to aggregators, then context attributes to context widgets, and finally identify the need for interpreters to abstract or infer context information.

According to Figure 16, our model for the Conference Assistant application consists of the following entities: Attendee, Presenter, Room, Presentation, Slide, and Web page. Each of these entities has a number of attributes, also presented in Figure 16. According to our mapping rules, the software design of the application contains one aggregator component for each entity, and one context widget for each attribute of an entity. We apply them to produce the aggregators and widgets sections of the software components list shown in Figure 17.

In the Conference Assistant application, we do not use complex context inference. However, we need interpreters to abstract information acquired from our iButton and/or PinPoint sensors. We assume that we may use either of the two sensing technologies during the life cycle of the application. As a consequence, we need interpreters on one hand to transform an iButton ID to an attendee name and on the other hand, to transform both tag ID and zone ID provided by the PinPoint system to attendee and room names.

We now show how the list of components shown in Figure 17 can be simplified, and we examine the tradeoffs involved. First, let us consider aggregators that provide a single piece of context information acquired directly through a context widget, like Presenter, Room, and Presentation. Because aggregators are intended to aggregate information from multiple sources, an aggregator with only one source of information is of limited usefulness. Such an aggregator can be removed from the software design. The gain is mostly in simplicity of implementation and performance. The main drawback concerns future extensibility. If new context attributes of the Presentation entity, such as the number of attendees and their overall interest were to be added, the aggregator would make these new attributes easier to access. As a result, we can remove the Presenter, Room, and Presentation Aggregators and let the application interact directly with their respective widgets. If attributes of an en-

Figure 17. **Software components derived for the Conference Assistant.**

- Aggregators
  - Attendee
  - Presenter
  - Room
  - Presentation
  - Slide
  - Web page
- Widgets
  - Name (Attendee)
  - Room (Attendee)
  - Times of departure/arrival (Attendee)
  - Level of interest (Attendee)
  - Name (Presenter)
  - Name (Room)
  - Room (Presentation)
  - Number (Slide)
  - Title (Slide)
  - Thumbnail (Slide)
  - URL (Web page)
  - Title (Web page)
  - Thumbnail (Web page)
  - Current slide (Question)
  - Time (Question)
  - Current slide (User note)
  - Time (User note)
- Interpreters
  - Tag ID to Name (Attendee)
  - Zone ID to Room (Attendee)
  - iButton ID to Name (Attendee)

Note. Entity name in parentheses indicates the entity to which the attribute relates.

tity are clearly static throughout the expected usage period of the application, they do not need to be acquired through context widgets. As a result, the name widget associated with the Room Aggregator can also be removed, because the room name does not change.

8.5. From Model to Reality: The Conference Assistant Implementation

The actual Conference Assistant application was developed before we articulated the design method we just followed. This application was one of several that helped us gain experience with the design of context-aware applications. It is interesting to look at the actual implementation of the Conference Assistant and compare it to the design produced by our approach. Our original design produced a slightly more compact architecture, at the expense of modularity.

The main differences between the software design presented in the previous paragraph and the actual design of the Conference Assistant application are due to factoring. In the actual implementation, two kinds of factoring have occurred. First, entities that are very similar, namely the Slide and Web page entities, have been factored into a single widget. Second, logically related entities, or context attributes, have been implemented as a single aggregator or widget. For example, a single aggregator, called Presentation, implements both the Presentation entity and the result of the factoring of Slide and Web page. Similarly, context attributes of a Slide or Web page are acquired by a single widget called Content. For the Attendee entity, context attributes have been factored according to the moment when they are acquired. Contact information and interests are acquired at registration time by a registration widget. The room that the attendee enters or leaves as well as the corresponding times are acquired by a location widget. As a result, the overall software design is made up of fewer components, as shown in Figure 18. Modularity has been traded off for simplicity of implementation. Figure 19 shows the implemented software architecture of the Conference Assistant application.

A prototype of the Conference Assistant application has been implemented, deployed, and tested in our laboratory, with three presentation rooms and a few handheld devices. Before deploying this in a real conference environment, the scalability of the implemented architecture needs to be tested. In addition, a great obstacle to real deployment is the large infrastructure requirement both for detecting indoor location, providing ubiquitous communications, and supplying handheld devices to conference attendees. However, the application is both interesting and complex in terms of the context it uses and was useful for testing the context toolkit and our ideas about how to design context-aware applications.

With this example application and design, we have highlighted the major benefits of our approach. We break down the design into a series of manageable steps whose outcome is clearly defined. Along the way, the context abstractions shield application designers from the irrelevant technical details of the underlying architecture. They also map to high-level components that facilitate the description of complex applications.

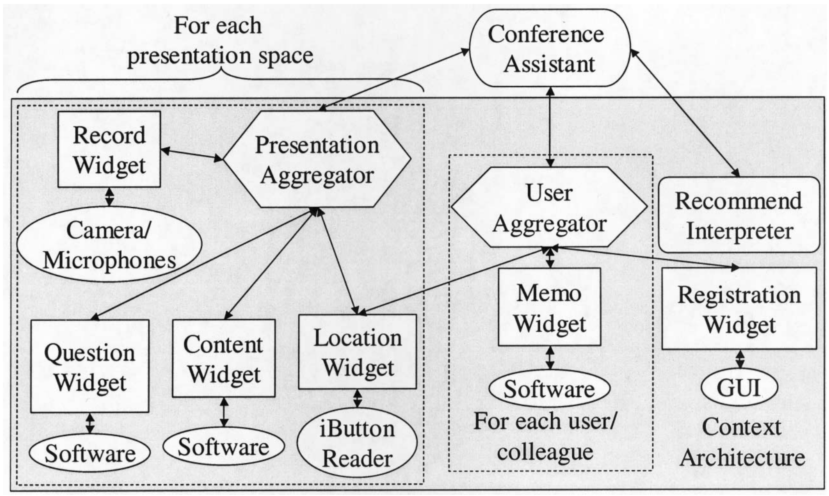
9. APPLICATIONS SUMMARY

In the previous sections, we described five applications that we built with the Context Toolkit to help us explore the space of context-aware applications. The applications used standard context such as location and identity,

***Figure 18.* Actual components and responsibilities in the Conference Assistant implementation.**

Component	Type	Responsibility
Attendee	Aggregator	Aggregates information about a user
Presentation	Aggregator	Aggregates information about a presentation
Registration	Widget	Acquires contact info, interests, and colleagues
Location	Widget	Acquires location and arrivals/departures of users
Content	Widget	Acquires title, URL/slide number, and thumbnail from slide or Web page
Question	Widget	Acquires audience questions and relevant presentation info
Memo	Widget	Acquires user's notes and relevant presentation info
Record	Widget	Acquires audiovisual presentation info
Name	Interpreter	Transforms iButton ID into user name

***Figure 19.* Conference Assistant context architecture.**



but have also used time and user activity as well. The applications supported all three types of context-aware behaviors: displaying context (In/Out Board, DUMMBO, the Intercom, and the Conference Assistant), automatically executing a service based on available context (In/Out Board, Context-Aware Mailing List, DUMMBO, the Intercom and the Conference Assistant), and tagging captured information with context for later retrieval (DUMMBO and the Conference Assistant).

With these applications, we described how the conceptual framework and our implementation of it, the Context Toolkit, can facilitate the design and implementation of context-aware applications. In particular, we used the Conference Assistant application to describe how to identify context components from the application requirements and to construct a software design. We demonstrated how context-aware applications can reuse abstraction components in the toolkit, how applications can be evolved to use different underlying sensing technology and to change their use of context, and how context-aware applications can acquire and make use of complex forms of context in useful ways. In the next section, we discuss related architectures and middleware that can be used to support the building of context-aware applications.

10. RELATED FRAMEWORKS

Over the past few years, there have been a number of frameworks and pieces of middleware developed to support different aspects of ubiquitous computing. Some were developed specifically to deal with context-aware computing, whereas others were more generic in their focus. In general, these systems do not address all of the issues we have presented in terms of acquiring, handling, and taking action on context. We learned from these systems and, as a result, constructed our own conceptual framework and implementation. In this section, we discuss some of these related systems.

10.1. Context-Aware Frameworks

Several frameworks have been built or proposed for supporting context-aware applications. These frameworks have been influential and complementary to our research but do not provide complete support for the set of required features that we presented in Section 3.1.

Schilit's System Architecture. In his Ph.D. thesis, Schilit (1995) presented a system architecture that supported context-aware mobile computing. This architecture was the first to support context-aware computing and has been very influential to our own research. It has helped us to identify the important features of context and context awareness and to identify some of the difficult problems in building context-aware applications. Schilit's work was focused on making context-aware computing applications *possible* to build. Our work, instead, focuses on making these applications *easier* to build. This difference in focus begins to delineate where our research differs. Schilit's architecture supported the gathering of context about devices and users. He had three main components in his system: (a) device agents that maintain status and capabilities of devices, (b) user agents that maintain user preferences, and (c) active maps that maintain the location information of devices and users (Schilit & Theimer, 1994). The architecture did not support or provide guidelines for the acquisition of context. Instead device and user agents were built on an individual basis, tailored to the set of sensors that each used. This makes it very difficult to evolve existing applications, both in changing how applications use context and in changing the set of underlying sensors. In addition, this architecture did not support context interpretation or storage, leaving applications to implement these features.

Stick-e Notes. The Stick-e Notes system is a general framework for supporting nonprogrammers in building context-aware applications (Brown, 1996). Whereas our research is looking at supporting the acquisition and delivery of context, this research is complementary to ours and focuses on how to support application designers in actually using the context to perform context-aware behaviors. It provides a general mechanism for indicating what context an application designer wants to use and provides simple semantics for writing rules, or Stick-e Notes, that specify what actions to take when a particular combination of context is realized. A group of notes or rules are collected together to form a Stick-e document. A context-aware application consists of the document and the Stick-e note architecture.

CyberDesk. In our previous research on context-aware computing, we built an architecture called CyberDesk (Dey et al., 1998). This architecture was built to automatically integrate Web-based services based on virtual context, or context derived from the electronic world. The virtual context was the personal information the user was interacting with on-screen including e-mail addresses, mailing addresses, dates, names, and so forth. The virtual context was used to infer what activity the user was involved in, in order to display a list of relevant Web-based services. Although it was limited in the types of context it could handle, CyberDesk contained many of the mechanisms that we believe are necessary for a general context-aware architecture. Applications simply specified what context types they were interested in, and were notified when those context types were available. The modular architecture supported automatic interpretation—that is, automatically interpreting individual and multiple pieces of context to produce an entirely new set of derived context. The architecture also supported the abstraction of context information and aggregation/combination of context information. We moved away from this architecture because it did not support multiple simultaneous applications, used a centralized mechanism, and did not support querying or storage of context.

Context And Location Aware Information Service (CALAIS). CALAIS was another architecture that was designed to support context-aware applications (Nelson, 1998). This work was performed to solve two problems: the ad hoc nature of sensor use and the lack of a fine-grained location information management system. An abstraction was developed to hide the details of sensors from context-aware applications. However, similar to Schilit's architecture, there was very little support to aid developers in adding new sensors to the architecture and the architecture did not support storage of context or interpretation of context, leaving application developers to provide their own mechanisms on an individual basis. CALAIS supported the use of distributed context sensing and provided query and notification mechanisms. An interesting feature in this work was the use of composite events, being able to subscribe to a combination of events. For example, an application could request to be notified when Event B occurred after Event A with no intervening events. This is a powerful mechanism that makes the acquisition and analysis of context easier for application developers.

CoolTown. CoolTown is an infrastructure that supports context-aware applications by representing each real-world object, including people, places and devices, with a Web page (Caswell & Debaty, 2000). Each Web page dynamically updates itself as it gathers new information about the entity that it represents, similar to our aggregator abstraction. CoolTown is primarily aimed at supporting applications that display context and services to end-users. For example, as a user moves throughout an environment, they will see a list of available services for this environment. They can request additional information or can execute one of the services. The CoolTown infrastructure provides abstraction components (URLs for sensed information and Web pages for entities) and a discovery mechanism to make the building of these types of applications easier. However, it was not designed to support interpretation of low-level sensed information. As well, although it focuses on displaying context, it was not intended to support other context-aware features of automatically executing a service based on context or tagging captured information with context.

Situated Computing Service. The Situated Computing Service is a proposed architecture that is similar to CyberDesk for supporting context-aware applications (Hull et al., 1997). It insulates applications from sensors used to acquire context. A Situated Computing Service is a single server that is responsible for both context acquisition and abstraction. It provides both querying and notification mechanisms for accessing relevant information. A single prototype server has been constructed as proof of concept, using only a single sensor type, so its success is difficult to gauge.

The Situated Computing Service provides no support for acquiring sensor information, only delivering it.

Context Information Service (CIS). The is another proposed architecture for supporting context-aware applications (Pascoe, 1998). It has yet to be implemented at any level but contains some interesting features. It supports the interpretation of context and the choosing of a sensor to provide context information based on a quality of service guarantee. In contrast to the Situated Computing Service, it promotes a tight connection between applications and the underlying sensors, taking an application-dependent approach to system building. The CIS maintains an object-oriented model of the world where each real-world object is represented by an object that has a set of predefined states. Objects can be linked to each other through relations such as "close to." For example, the set of nearby printers would be specified by a "close to" relation with a user, a given range, and "printers" as the candidate object. The set would be dynamically updated as the user moves through an environment.

10.2. Relevant Middleware

The previously described infrastructures, including the Context Toolkit, contain specific components that support the acquisition and handling of context. There are a number of pieces of middleware that, although not addressing context, address some similar issues of supporting multiple applications, multiple users, multiple devices, and multiple sources of data. We discuss some of these middleware approaches here—the use of abstractions, agents, and tuple spaces.

Abstractions. Both the AROMA project (Pederson & Sokoler, 1997) and the metaDesk system (Ullmer & Ishii, 1997) provided widget-like abstractions for dealing with physical sensors. The AROMA project attempted to provide peripheral awareness of remote colleagues through the use of abstract information. Its object-oriented architecture used the concepts of sensor abstraction and interpretation. Playing the role of the application were synthesizers that take the abstract awareness information and display it. It did not provide any support for adding new sensors or context types, although sensor abstraction made it easier to replace sensors. The metaDesk system was a platform for demonstrating tangible user interfaces. Instead of using GUI widgets to interact with the system, users used physical icons to manipulate information in the virtual world. The system used sensor proxies to separate the details of individual sensors from the application. This system architecture supported distribution and a namespaces mechanism to allow simple runtime evolution of applications. Interpretation of the sensed information was left to individual applications.

Agents. The Adaptive Agent Architecture (Kumar, Cohen, & Levesque, 2000), Hive (Minar, Gray, Roup, Krikorian, & Maes, 2000), and MetaGlue (Coen, Philips, Warshawshy, & Weisman, 1999) are all agent-based middleware for supporting multimodal applications and ubiquitous computing environments. Agents are abstractions used to represent sensors and services. In the Adaptive Agent Architecture, when a sensor has data available, the agent representing it places the data in a centralized blackboard. When an application needs to handle some user input, the agent representing it translates the input and places the results on the blackboard. Applications indicate what information they can use through their agents. When useful data appear on the blackboard, the relevant applications' agents are notified and these agents pass the data to their respective applications. The blackboard provides a level of indirection between the applications and sensors, effectively hiding the details of the sensors. Similarly, Hive and MetaGlue use mobile agents to acquire information from the environment and distribute it to applications. The agent-based approach could be used to implement our conceptual framework, but a number of concepts would have to be added, including the ability to interpret, aggregate, and store context information.

Tuple Spaces. Limbo (Davies et al., 1997), JavaSpaces (Sun Microsystems, 2000), and T spaces (Wyckoff, McLaughry, Lehman, & Ford, 1998) are all tuple-space-based infrastructures based on the original Linda tuple space (Gerlernter, 1985). Tuple spaces offer a centralized or distributed blackboard into which items, or tuples, can be placed and retrieved. Tuple spaces provide an abstraction layer between components that place tuples and components that retrieve tuples. Although this abstraction meets our separation of concerns requirement, it does not provide support for interpretation, aggregation, or persistent context storage, essential requirements for context-aware computing.

11. USING THE CONCEPTUAL FRAMEWORK AS A RESEARCH TESTBED

In Section 4.6, we described how we used the Context Toolkit to design and implement context-aware applications. Although the toolkit addresses a lot of the basic requirements of context-aware applications, there are still a number of research issues to address to facilitate the design and development of these types of applications. We have used the Context Toolkit as a research testbed to explore a number of these issues, including the representation and acquisition of context, privacy, ambiguity in sensed data, and higher level programming abstractions.

11.1. Representation and Acquisition of Context

The categories we presented in Section 2.2 provide a rough but useful breakdown of context information. A finer taxonomy of context information would serve two major purposes: It would better support design by providing a checklist of context an application may need to acquire and use, and it would support easier implementation by allowing developers to provide a standardized library of context components. To pursue a parallel with GUIs, most GUI applications today can take advantage of standard components that encapsulate complex interactions with the user. Similarly, context-aware applications could rely on standard components that provide context information about their environment.

To achieve this in a general and usable way, however, and to provide more interesting and useful applications than simple rule-based automata, we must expect to have to tackle difficult issues of knowledge modeling and representation. The uses of context we envision require that applications be capable of maintaining a basic dynamic model of the real world. This includes both a model of the environment, as suggested by Harter, Hopper, Steggles, Ward, and Webster (1999) and Brumitt, Shafer, Krumm, and Meyers (1999), and more detailed models of the entities within that environment. To be able, for example, to signal to the user that she should stop for milk while passing by the grocery store, or to suggest an interesting activity on an out-of-town weekend, requires that accumulated knowledge about the user's interests and whereabouts, as well as her social situation, be available and meaningful to the application. This requires moving beyond the environmental context used in traditional context-aware research to include more general context that cannot be sensed directly from the environment (e.g., user interests, shared knowledge and social relationships between users in the environment, and user interruptibility). Although the Context Toolkit currently does not address this type of context, it provides support for acquiring and delivering it when it does become available.

11.2. Privacy

As computational capabilities seep into our environments, there is an ever-growing and legitimate concern that technologists are not spending enough of their intellectual cycles on the social implications of their work. There is hope, however. Context-aware computing holds a promise of providing mechanisms to support some social concerns. For example, using context to tag captured information may at first seem like an intrusion to individual privacy ("I do not want to meet in a room that records what I say."). However, that same context can be used to protect the concerns of individuals by providing computational support to establish default access control lists to any information that has been captured (Lau, Etzioni, & Weld, 1999), limiting distribution to those who were in attendance.

We took this approach in our initial exploration of handling privacy concerns (Nguyen, Tullio, Drewes, & Mynatt, 2000). When any context is collected by a context widget, it can be assigned an owner. The owner specifies a set of rules indicating what context can be seen by others and under what situations. For example, you may provide a family member with complete access to your schedule but only allow a colleague to view your schedule during working hours and only allow strangers to see what times you are not busy. Now, when any component or application wants to access context from a widget, it must first authenticate itself using standard public–private key encryption. If the component is authorized for full access to the context, according to the context owner's rules, the context is delivered as usual. However, if the component is not authorized or only partially authorized for access to this context, no information or partial information is delivered. This is not a complete solution, obviously, but holds the potential to be further explored with the Context Toolkit.

11.3. Ambiguity in Context Data

An assumption that we have made in our application development is that the context we are sensing is 100% accurate. Obviously, this is a false assumption because there is often a difference between the actual context and the context that we can sense. There is a real need to deal with this difference, which results in inaccurate or ambiguous context. There are three complementary approaches for addressing this issue: passing ambiguity on to applications, attempting to disambiguate context automatically, and attempting to disambiguate context manually. An application can specify the accuracy of the context it needs to the context-sensing infrastructure. If the infrastructure can supply the necessary context with the specified accuracy, then it provides the context along with a measure of how accurate it is. If the infrastructure is unable to meet the accuracy requirements, it can notify the application of its available accuracy and can allow the application to decide on a future course of action (e.g., request the context with a lower degree of accuracy or disregard that context).

A second approach is to use data from multiple sensors, if available, to improve the accuracy of the available context. Multiple homogeneous sensors can be used to remove the noise in sensing techniques. Multiple heterogeneous sensors can be used to offset the failures that exist in each other. For example, computer vision tends to work very poorly in low lighting conditions, but is not affected by electromagnetic interference in the environment. RF-based sensors operate at the same level of accuracy in any lighting conditions, but are susceptible to electromagnetic interference. The process of using multiple sensors to disambiguate sensor information or context is called *sensor fusion* (Brooks & Iyengar, 1997). Of course, there are limits to what we can currently acquire from sensors and what we can infer from them. There is also a gap between our ability to sense and infer and sensor fusion cannot completely alleviate this problem.

A third approach for dealing with ambiguous or inaccurate sensor data, and the approach that we have explored, is to provide an interface that allows a user to manually disambiguate it (Dey, Mankoff, & Abowd, 2000; Mankoff, Abowd, & Hudson, 2000). Rather than hiding from the user the fact that the application is using inaccurate data, the application can instead notify the user that there are multiple possible alternatives for interpreting the sensor data. If the user chooses to ignore the ambiguity, the application can choose a default alternative. If the user chooses to actively select one of the alternatives, the application can proceed with the chosen alternative. In addition, it must notify the context-sensing architecture of the chosen alternative to allow other applications that use the same context information to take advantage of the manually disambiguated context.

11.4. Higher Level Programming Abstractions

The Context Toolkit allows a programmer to model a world instrumented with sensors. Context widgets map to sensors and context aggregators map to people, places, and objects in the real world. Although the model matches the real world quite well, it forces programmers to think about and design applications in terms of these individual component abstractions. Instead of this, perhaps we can allow programmers to use higher level abstractions that sit on top of the component abstractions. The higher the abstraction, the easier it should be for the programmer to use, at the expense of having some information temporarily hidden from the programmer.

We have investigated the use of situations as our higher level programming abstraction (Dey & Abowd, 2000a). A *situation* is a collection of context data that may or may not be accessed from an individual component in the Context Toolkit. For example, a user may want to access several pieces of context about an individual. This information can easily be acquired from that individual's aggregator. However, a user may want to know when his stock broker arrives in his office, whether the share price of a particular stock has risen above a certain value, and if the user has some free time to call his broker. This information cannot be acquired from a single component but must be acquired from multiple components and then analyzed to determine when the situation occurred. Typically, these steps are left to the application programmer to perform. However, by allowing a programmer to specify a complex situation (i.e., a real-world event), and having the infrastructure automatically collect and analyze the context from multiple components, we can make the programmer's job easier.

12. CONCLUSIONS

Context-aware computing is still in its infancy. We have attempted to clarify the notion of context and laid out foundations for the design and development of context-aware applications. First, we defined broad categories of entities capable of providing context information and identified basic categories of context. Using separation of concerns as a guiding software engineering principle, we built on this classification to introduce abstractions that encapsulate common context operations. These abstractions form a conceptual framework for supporting context-aware applications and map directly to components in our Context Toolkit. Using a variety of applications that we have implemented, we demonstrated how the Context Toolkit can be used to design and build context-aware applications and how it supports reusability of components, evolution of applications, and the acquisition and use of complex context. We introduced a structured design approach that helps with identifying context abstractions and leads to a software design. We demonstrated this approach with the design of a complex context-aware application intended for conference attendees. Finally, we outlined research issues that we are exploring with the Context Toolkit to enable further and more elaborate and realistic context-aware applications.

Issues dealing with the use and impact of context-aware computing go beyond the social problems raised earlier. Ultimately, it will be the authentic experience with context-aware computing in our everyday lives that will help us to learn how it impacts individuals and groups. The main contribution of the context-aware toolkit infrastructure we presented in this anchor article should be seen as a bootstrapping mechanism to encourage others more creative than us to explore this emerging applications area. In the end, it is not one killer *application* that will cause the widespread use of context-aware computing. Rather, it is the killer *existence* that we should be able to demonstrate through the coordinated and intelligent interaction of our everyday activities and computational services.

NOTES

Background. This article is based on Anind K. Dey's doctoral thesis. For more information on the Context Toolkit described in this article, including gaining access to it for further research and development, please contact us and visit our Web page: http://www.cc.gatech.edu/fce/contexttoolkit/.

Acknowledgments. We thank members of the Future Computing Environments (FCE) Group at Georgia Tech for their continued influence on the development of our ideas on context-aware computing. In particular, we thank Jennifer Mankoff for her original insights that lead to the analogy with GUI widgets and context.

Support. This work is supported in part by grants from the National Science Foundation (Career Grant IRI–9703384 and Experimental Systems Software Grant EIA–9806822) and from the University Partnerships in Research program at Motorola through sponsorship from Dr. Ron Borgstahl and Dr. Ken Crisler.

Authors' Present Addresses. Anind K. Dey, Intel Research, 2150 Shattuck Avenue, Penthouse Suite, Berkeley, CA 94704. E-mail: anind@intel-research.net. Gregory D. Abowd, College of Computing and GVU Center, Georgia Institute of Technology, Atlanta, GA 30332–0280. E-mail: abowd@cc.gatech.edu. Daniel Salber, IBM T.J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532. E-mail: salber@acm.org.

HCI Editorial Record. First manuscript received January 18, 2000. Revision received October 16, 2000. Accepted by Thomas Moran. Final manuscript received January 8, 2001. — *Editor*

REFERENCES

Abowd, G. D., Atkeson, C. G., Hong, J., Long, S., Kooper, R., & Pinkerton, M. (1997). Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks, 5,* 421–433.

Arons, B. (1991). The design of audio servers and toolkits for supporting speech in the user interface. *Journal of the American Voice I/O Society*, 9, 27–41.

Bauer, M., Heiber, T., Kortuem, G., & Segall, Z. (1998). A collaborative wearable system with remote sensing. *Proceedings of the 2nd International Symposium on Wearable Computers (ISWC 98).* Los Alamitos, CA: IEEE.

Bederson, B. B. (1995). Audio augmented reality: A prototype automated tour guide. *Proceedings of the CHI 95 Conference on Human Factors in Computing Systems.* New York: ACM Press.

Beigl, M. (2000). MemoClip: A location based remembrance appliance. *Personal Technologies*, *4*(4), 230–234.

Brooks, R. R., & Iyengar, S. S. (1997). *Multi-sensor fusion: Fundamentals and applications with software.* Englewood Cliffs, NJ: Prentice Hall.

Brotherton, J., Abowd, G. D., & Truong, K. (1999). *Supporting capture and access interfaces for informal and opportunistic meetings* (GIT–GVU–99–06). Atlanta: Georgia Institute of Technology, GVU Center.

Brown, P. J. (1996). The Stick-e document: A framework for creating context-aware applications. *Proceedings of Electronic Publishing 96.* Laxenburg, Austria: IFIP.

Brown, P. J., Bovey, J. D., & Chen, X. (1997). Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications, 4*(5), 58–64.

Brumitt, B. L., Shafer, S., Krumm, J., & Meyers, B. (1999). *EasyLiving and the role of geometry in ubiquitous computing.* Paper presented at the DARPA/NIST/NSF Workshop on Research Issues in Smart Computing Environments, Atlanta, GA.

Caswell, D., & Debaty, P. (2000). Creating Web representations for places. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K).* Heidelberg, Germany: Springer-Verlag.

Coen, M., Philips, B., Warshawshy, N., & Weisman, L. (1999). Meeting the computational needs of intelligent environments: The MetaGlue environment. *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE 99).* Heidelberg, Germany: Springer-Verlag.

Dallas Semiconductor. (1999). *iButton home page* [On-line]. Available: http://www.ibutton.com/

Davies, N., Mitchell, K., Cheverst, K., & Blair, G. (1998). Developing a context-sensitive tour guide (GIST Technical Report G98–1). *Proceedings of the 1st Workshop on Human Computer Interaction for Mobile Devices.* Glasgow, Scotland: University of Glasgow.

Davies, N., Wade, S., Friday, A., & Blair, G. (1997). Limbo: A tuple space based platform for adaptive mobile applications. *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP 97).* Toronto, Canada.

Dey, A. K. (1998). Context-aware computing: The CyberDesk project. *Proceedings of the AAAI 98 Spring Symposium on Intelligent Environments.* Menlo Park, CA: AAAI Press.

Dey, A. K. (2000). *Providing architectural support for building context-aware applications.* Unpublished doctoral thesis, Georgia Institute of Technology, Atlanta.

Dey, A. K., & Abowd, G. D. (2000a). CybreMinder: A context-aware system for supporting reminders. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K).* Heidelberg, Germany: Springer-Verlag.

Dey, A. K., & Abowd, G. D. (2000b). Towards a better understanding of context and context- awareness. *Proceedings of the What, Who, Where, When, and How of Context-Awareness Workshop, CHI 2000 Conference on Human Factors in Computer Systems.* New York: ACM.

Dey, A. K., Abowd, G. D., & Wood, A. (1998). CyberDesk: A framework for providing self–integrating context–aware services. *Knowledge Based Systems, 11*(1), 3–13.

Dey, A. K., Futakawa, M., Salber, D., & Abowd, G. D. (1999). The Conference Assistant: Combining context-awareness with wearable computing. *Proceedings of the 3rd International Symposium on Wearable Computers (ISWC 99).* Los Alamitos, CA: IEEE.

Dey, A. K., Mankoff, J., & Abowd, G. D. (2000). *Distributed mediation of imperfectly sensed context in ubiquitous computing environments* (GIT–GVU–00–14). Atlanta: Georgia Institute of Technology, GVU Center.

Dey, A. K., Salber, D., & Abowd, G. D. (1999). A context-based infrastructure for smart environments. *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE 99).* Heidelberg, Germany: Springer-Verlag.

Dourish, P., Edwards, W. K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Thornton, J., & Terry., D. B. (2000). Extending document management systems with active properties. *ACM Transactions on Information Systems, 18*, 140–170.

Feiner, S., MacIntyre, B., Hollerer, T., & Webster, T. (1997). A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies, 1,* 208–217.

Fels, S., Sumi, Y., Etani, T., Simonet, N., Kobayshi, K., & Mase, K. (1998). Progress of C–MAP: A context-aware mobile assistant. *Proceedings of the AAAI 98 Spring Symposium on Intelligent Environments.* Menlo Park, CA: AAAI Press.

Fox, A. (1998). *A framework for separating server scalability and availability from Internet application functionality.* Unpublished doctoral dissertation, University of California, Berkeley.

Franklin, D., & Flaschbart, J. (1998). All gadget and no representation makes jack a dull environment. *Proceedings of the AAAI 98 Spring Symposium on Intelligent Environments.* Menlo Park, CA: AAAI Press.

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

Harrison, B. L., Fishkin, K. P., Gujar, A., Mochon, C., & Want, R. (1998). Squeeze me, hold me, tilt me! An exploration of manipulative user interfaces. *Proceedings of the CHI 98 Conference on Human Factors in Computer Systems.* New York: ACM.

Harter, A., Hopper, A., Steggles, P., Ward, A., & Webster, P. (1999). The anatomy of a context- aware application. *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom 99).* New York: ACM.

Healey, J., & Picard, R. W. (1998). StartleCam: A cybernetic wearable camera. *Proceedings of the 2nd International Symposium on Wearable Computers (ISWC 98).* Los Alamitos, CA: IEEE.

Heiner, J. M., Hudson, S. E., & Tanaka, K. (1999). The information percolator: Ambient information display in a decorative object. *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST 96).* New York: ACM.

Hertz. (1999). *NeverLost* [On-line]. Available: http://www.hertz.com/serv/us/prod_lost.html

Hinckley, K., Pierce, J., Sinclair, M., & Horvitz, E. (2000). Sensing techniques for mobile interaction. *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST 2000).* New York: ACM.

Hull, R., Neaves, P., & Bedford-Roberts, J. (1997). Towards situated computing. *Proceedings of the 1st International Symposium on Wearable Computers (ISWC 97).* Los Alamitos, CA: IEEE.

IBM. (2000). *IBM Voice Systems home page* [On-line]. Available: http://www.ibm.com/software/speech/

Ishii, H., & Ullmer, B. (1997). Tangible bits: Towards seamless interfaces between people, bits and atoms. *Proceedings of the CHI 97 Conference on Human Factors in Computing Systems.* New York: ACM.

Kiciman, E., & Fox, A. (2000). Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K).* Heidelberg, Germany: Springer-Verlag.

Kidd, C. D., O'Connell, T., Nagel, K., Patil, S., & Abowd, G. D. (2001). *Building a better intercom: Context-mediated communication within the home* (GIT–GVU–00–27). Atlanta: Georgia Institute of Technology, GVU Center.

Kumar, S., Cohen, P. R., & Levesque, H. J. (2000). The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS 2000).* Los Alamitos, CA: IEEE.

Lamming, M., & Flynn, M. (1994). Forget-me-not: Intimate computing in support of human memory. *Proceedings of the FRIEND 21: International Symposium on Next Generation Human Interfaces.* Tokyo, Japan: Institute for Personalized Information Environment.

Lau, T., Etzioni, O., & Weld, D. S. (1999). Privacy interfaces for information management. *Communications of the ACM, 42*(10), 88–94.

MacIntyre, B., & Feiner, S. (1996). Language-level support for exploratory programming of distributed virtual environments. *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST 96).* New York: ACM.

Mankoff, J., Abowd, G. D., & Hudson, S. (2000). Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. *Proceedings of the CHI 2000 Conference on Human Factors in Computer Systems.* New York: ACM.

Marmasse, N., & Schmandt, C. (2000). Location aware information delivery with comMotion. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K).* Heidelberg, Germany: Springer-Verlag.

McCarthy, J. F., & Anagost, T. D. (2000). EventManager: Support for the peripheral awareness of events. *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K).* Heidelberg, Germany: Springer-Verlag.

McCarthy, J. F, & Meidel, E. S. (1999). ActiveMap: A visualization tool for location awareness to support informal interactions. *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC 99).* Heidelberg, Germany: Springer-Verlag.

Minar, N., Gray, M., Roup, O., Krikorian, R., & Maes, P. (2000). Hive: Distributed agents for networking things. *IEEE Concurrency, 8*(2), 24–33.

Mynatt, E. D., Back, M., Want, R., Baer, M., & Ellis, J. B. (1998). Designing audio aura. *Proceedings of the CHI 98 Conference on Human Factors in Computing Systems.* New York: ACM.

Nelson, G. J. (1998). *Context-aware and location systems.* Unpublished doctoral dissertation, University of Cambridge, Cambridge, England.

Nguyen, D., Tullio, J., Drewes, T., & Mynatt, E. (2000). *Dynamic door displays* [On-line]. Available: http://ebirah.cc.gatech.edu/~jtullio/doorshort.htm

Pascoe, J. (1998). Adding generic contextual capabilities to wearable computers. *Proceedings of the 2nd International Symposium on Wearable Computers (ISWC 98).* Los Alamitos, CA: IEEE.

Pascoe, J., Ryan, N. S., & Morse, D. R. (1998). Human–computer–giraffe Interaction: HCI in the field. *Proceedings of the Workshop on Human Computer Interaction with Mobile Devices.* GIST Technical Report G98–1. Glasgow, Scotland: University of Glasgow.

Pederson, E. R., & Sokoler, T. (1997). AROMA: Abstract representation of presence supporting mutual awareness. *Proceedings of the CHI 97 Conference on Human Factors in Computing Systems.* New York: ACM.

PinPoint. (1999). *PinPoint 3D–iD introduction* [On-line]. Available: http://www.pinpointco.com/products/products_title.htm

Rekimoto, J. (1996). Tilting operations for small screen interfaces. *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST 96).* New York: ACM.

Rekimoto, J. (1999). Time-machine computing: A time-centric approach for the information environment. *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST 99).* New York: ACM.

Rhodes, B. J. (1997). The wearable remembrance agent. *Proceedings of the 1st International Symposium on Wearable Computers (ISWC 97).* Los Alamitos, CA: IEEE Press.

Rodden, T., Cheverst, K., Davies, K., & Dix, A. (1998). Exploiting context in HCI design for mobile systems. *Proceedings of the Workshop on Human Computer Interaction with Mobile Devices.* Glasgow, Scotland.

Ryan, N., Pascoe, J., & Morse, D. (1997). Enhanced reality fieldwork: The context-aware archaeological assistant. In V. Gaffney, M. V. Leusen, & S. Exxon (Eds.), *Computer applications in archaeology.* Oxford, United Kingdom: Tempus Reparatum.

Salber, D., Dey, A. K., & Abowd, G. D. (1999). The context toolkit: Aiding the development of context-enabled applications. *Proceedings of the CHI 99 Conference on Human Factors in Computer Systems.* New York: ACM.

Schilit, B. (1995). *System architecture for context-aware mobile computing.* Unpublished doctoral dissertation, Columbia University, New York.

Schilit, B., Adams, N., & Want, R. (1994). Context-aware computing applications. *Proceedings of the 1st International Workshop on Mobile Computing Systems and Applications.* Los Alamitos, CA: IEEE.

Schilit, B., & Theimer, M. (1994). Disseminating active map information to mobile hosts. *IEEE Network, 8*(5), 22–32.

Schmidt, A., Aidoo, K. A., Takaluoma, A., Tuomela, U., Laerhoven, K. V., & Velde, W. V. d. (1999). Advanced interaction in context. *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC 99).* Heidelberg, Germany: Springer-Verlag.

Schmidt, A., Takaluoma, A., & Mäntyjärvi, J. (2000). Context-aware telephony over WAP. *Personal Technologies, 4,* 225–229.

Schwartz, M. F., Emtage, A., Kahle, B., & Neuman, B. C. (1992). A comparison of Internet resource discovery approaches. *Computer Systems, 5,* 461–493.

Sun Microsystems. (1999). *Jini connection technology home page* [On-line]. Available: http://www.sun.com/jini/

Sun Microsystems. (2000). *JavaSpaces technology* [On-line]. Available: http://www.javasoft.com/products/javaspaces/index.html

SVRLOC Working Group of the IETF. (1999). *Service location protocol home page* [On-line]. Available: http://www.srvloc.org/

Ullmer, B., & Ishii, H. (1997). The metaDESK: Models and prototypes for tangible user interfaces. *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST 97).* New York: ACM.

Universal Plug and Play Forum. (2000). *Universal Plug and Play Forum home page* [On-line]. Available: http://www.upnp.org

Want, R., Hopper, A., Falcao, V., & Gibbons, J. (1992). The Active Badge location system. *ACM Transactions on Information Systems, 10*(1), 91–102.

Ward, A., Jones, A., & Hopper, A. (1997). A new location technique for the active office. *IEEE Personal Communications, 4*(5), 42–47.

Webster's new twentieth century dictionary of the English language. (1980). Springfield, MA: Merriam-Webster, Inc.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, *265*(3), 66–75.

Weiser, M., & Brown, J. S. (1997). The coming age of calm technology. In P. J. Denning & R. M. Metcalfe (Eds.), *Beyond calculation: The next fifty years of computing* (pp. 75–86). New York: Springer-Verlag.

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