Providing Architectural Support for Building Context-Aware Applications

A Thesis
Presented to
The Academic Faculty

by

Anind K. Dey

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
November 2000

Providing Architectural Support for Building Context-Aware Applications

Approved:

_____

Gregory D. Abowd, Chairman

_____

Mustaque Ahamad

_____

Blair MacIntyre

_____

Beth Mynatt

_____

Terry Winograd, External Advisor

Date Approved _____

DEDICATION

To my parents,

For all the times you started, but were never able to complete your PhDs

ACKNOWLEDGEMENTS

After four degrees, at two universities, in three different disciplines, I have learned one thing – I could never have done *any* of this, particularly the research and writing that went into this dissertation, without the support and encouragement of a lot of people.

First, I would like to thank my advisor, Gregory Abowd. I owe you so much. You've been my friend, my mentor, my confidant, my colleague, and a never-ending fount of moral support. You have given so much of yourself to help me succeed. If I do take the academic path, I only hope that I can be half the advisor that you have been to me. Whatever path I do take, I will be prepared because of you.

I would also like to thank the rest of my thesis committee for their support. Mustaque Ahamad, Blair MacInytre, Beth Mynatt and Terry Winograd provided me with invaluable advice and comments on both my research and my future research career plans.

I've been very lucky throughout most of my life in graduate school, in that I've been able to concentrate mostly on my research. This is due in a large part to the gracious support of Motorola and its University Partnership in Research (UPR) funding program. I would particularly like to thank Ron Borgstahl who initiated my UPR funding back in 1996 and supported me for over three years. I would also like to thank Ken Crisler from the Applications Research group at Motorola Labs.

I've also been fortunate to have a great group of friends at Georgia Tech. This includes my office mates in both the Multimedia Lab and in the CRB, the hardcore Happy Hour crew, and many other students and faculty, too numerous to name. Not only are you the people I can discuss my research with and goof off with, but also you are confidants who I can discuss my troubles with and who stand by me through thick and thin. This, I believe, is the key to getting through a Ph.D. program – having good friends to have fun with and complain to.

I would also like to express my thanks to my research group, both the larger Future Computing Environments group and the smaller Ubiquitous Computing group. I have learned so much from all of you, from figuring out what research is, to choosing a research agenda, to learning how to present my work. Your constructive criticism and collaboration have been tremendous assets throughout my Ph.D.

This work would not have been possible without the support of my best friend, Jennifer Mankoff. You're always there for me, when I need help with my research and when I need moral support. You were instrumental in helping me find my dissertation topic and in helping me get past all the self-doubting that inevitably crops up in the course of a Ph.D. You're the first person I turn to in good times and in bad. You have given me the courage to make the next transitions in my life. For all of this, I thank you.

Finally, I would like to dedicate this work to my family: Santosh, Prabha, Amitabh and Anjuli. Without your unending support and love from childhood to now, I never would have made it through this process or any of the tough times in my life. Thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Traditional interactive applications are limited to using only the input that users explicitly provide. As users move away from traditional desktop computing environments and move towards mobile and ubiquitous computing environments, there is a greater need for applications to leverage from implicit information, or context. These types of environments are rich in context, with users and devices moving around and computational services becoming available or disappearing over time. This information is usually not available to applications but can be useful in adapting the way in which it performs its services and in changing the available services. Applications that use context are known as context-aware applications.

This research in context-aware computing has focused on the development of a software architecture to support the building of context-aware applications. While developers have been able to build context-aware applications, they have been limited to using a small variety of sensors that provide only simple context such as identity and location. This dissertation presents a set of requirements and component abstractions for a conceptual supporting framework. The framework along with an identified design process makes it easier to acquire and deliver context to applications, and in turn, build more complex context-aware applications.

In addition, an implementation of the framework called the Context Toolkit is discussed, along with a number of context-aware applications that have been built with it. The applications illustrate how the toolkit is used in practice and allows an exploration of the design space of context-aware computing. This dissertation also shows how the Context Toolkit has been used as a research testbed, supporting the investigation of difficult problems in context-aware computing such as the building of high-level programming abstractions, dealing with ambiguous or inaccurate context data and controlling access to personal context.

CHAPTER 1

INTRODUCTION AND MOTIVATION

Humans are quite successful in conveying ideas to each other and reacting appropriately. This is due to many factors including the richness of the language they share, the common understanding of how the world works, and an implicit understanding of everyday situations. When humans talk with humans, they are able to use information apparent from the current situation, or *context*, to increase the conversational bandwidth. Unfortunately, this ability to convey ideas does not transfer well when humans interact with computers. Computers do not understand our language, do not understand how the world works and cannot sense information about the current situation, at least not as easily as most humans can. In traditional interactive computing, users have an impoverished mechanism for providing information to computers, typically using a keyboard and mouse. As a result, we must explicitly provide information to computers, producing an effect contrary to the promise of transparency in Weiser's vision of ubiquitous computing (Weiser 1991). We translate what we want to accomplish into specific minutiae on how to accomplish the task, and then use the keyboard and mouse to articulate these details to the computer so that it can execute our commands. This is nothing like our interaction with other humans. Consequently, computers are not currently enabled to take full advantage of the context of the human-computer dialogue. By improving the computer's access to context, we can increase the richness of communication in human-computer interaction and make it possible to produce more useful computational services.

Why is interacting with computers so different than interacting with humans? There are three problems, dealing with the three parts of the interaction: input, understanding of the input, and output. Computers cannot process and understand information as humans can. They cannot do more than what programmers have defined they are able to do and that limits their ability to understand our language and our activities. Our input to computers has to be very explicit so that they can handle it and determine what to do with it. After handling the input, computers display some form of output. They are much better at displaying their current state and providing feedback in ways that we understand. They are better at displaying output than handling input because they are able to leverage off of human abilities. A key reason for this is that humans have to provide input in a very sparse, non-conventional language whereas computers can provide output using rich images. Programmers have been striving to present information in the most intuitive ways to users, and the users have the ability to interpret a variety of information. Thus, arguably, the difficulty in interacting with computers stems mainly from the impoverished means of providing information to computers and the lack of computer understanding of this input. So, what can we do to improve our interaction with computers on these two fronts?

On the understanding issue, there is an entire body of research dedicated to improving computer understanding. Obviously, this is a far-reaching and difficult goal to achieve and will take time. The research we are proposing does not address computer understanding but attempts to improve human-computer interaction by providing richer input to computers.

Many research areas are attempting to address this input deficiency but they can mainly be seen in terms of two basic approaches:

- improving the language that humans can use to interact with computers; and,
- increasing the amount of situational information, or *context*, that is made available to computers.

The first approach tries to improve human-computer interaction by allowing the human to communicate in a much more natural way. This type of communication is still very explicit, in that the computer only knows what the user tells it. With natural input techniques like speech and gestures, no other information besides the explicit input is available to the computer. As we know from human-human interactions, situational information such as facial expressions, emotions, past and future events, the existence of other people in the room, and relationships to these other people are crucial to understanding what is occurring. The process of building this shared understanding is called grounding (Clark and Brennan 1991). Since both human participants in the interaction share this situational information, there is no need to make it explicit. However, this need for explicitness does exist in human-computer interactions, because the computer does not share this implicit situational information or context.

The two types of techniques (use of more natural input and use of context) are quite complementary. They are both trying to increase the richness of input from humans to computers. The first technique makes it easier to input explicit information while the second technique supports the use of unused implicit information that can be vital to understanding the explicit information. This thesis is primarily concerned with the second technique. We are attempting to use context as an implicit cue to enrich the impoverished interaction from humans to computers.

How do application developers provide the context to the computers, or make those applications aware and responsive to the full context of human-computer interaction and human-environmental interaction? We could require users explicitly to express all information relevant to a given situation. However, the goal of *context-aware computing*, or applications that use context, as well as computing in general, should be to make interacting with computers easier. Forcing users consciously to increase the amount of information they have to input would make this interaction more difficult and tedious. Furthermore, it is likely that most users will not know which information is potentially relevant and, therefore, will not know what information to provide.

We want to make it easier for users to interact with computers and the environment, not harder, by allowing users to not have to think consciously about using the computers. Weiser coined the term "calm technology" to describe an approach to ubiquitous computing, where computing moves back and forth between the center and periphery of the user's attention (Weiser and Brown 1997). To this end, our approach to context-aware application development is to collect implicit contextual information through automated means, make it easily available to a computer's run-time environment and let the application designer decide what information is relevant and how to deal with it. This is the better approach, for it removes the need for users to make all information explicit and it puts the decisions about what is relevant into the designer's hands. The application designer should have spent considerably more time analyzing the situations under which their application will be executed and can more appropriately determine what information could be relevant and how to react to it.

The need for context is even greater when we move into non-traditional, off-the-desktop computing environments. Mobile computing and ubiquitous computing have given users the expectation that they can access whatever information and services they want, whenever they want and wherever they are. With computers being used in such a wide variety of situations, interesting new problems arise and the need for context is clear: users are trying to obtain different information from the same services in different situations. Context can be used to help determine what information or services to make available or to bring to the forefront for users.

Applications that use context, whether on a desktop or in a mobile or ubiquitous computing environment, are called context-aware. The increased availability of commercial, off-the-shelf sensing technologies is making it more viable to sense context in a variety of environments. The prevalence of powerful, networked computers makes it possible to use these technologies and distribute the context to multiple applications, in a somewhat ubiquitous fashion. Mobile computing allows users to move throughout an

environment while carrying their computing power with them. Combining this with wireless communications allows users to have access to information and services not directly available on their portable computing device. The increase in mobility creates situations where the user's context, such as her location and the people and objects around her, is more dynamic. With ubiquitous computing, users move throughout an environment and interact with computer-enhanced objects within that environment. This also allows them to have access to remote information and services. With a wide range of possible user situations, we need to have a way for the services to adapt appropriately, in order to best support the human-computer and human-environment interactions. Context-aware applications are becoming more prevalent and can be found in the areas of wearable computing, mobile computing, robotics, adaptive and intelligent user interfaces, augmented reality, adaptive computing, intelligent environments and context-sensitive interfaces. It is not surprising that in most of these areas, the user is mobile and her context is changing rapidly.

We have motivated the need for context, both in improving the input ability of humans when interacting with computers in traditional settings and also in dynamic settings where the context of use is potentially changing rapidly. In the next section, we will provide a better definition of context and discuss our efforts in achieving a better understanding of context.

1.1 What is Context?

Realizing the need for context is only the first step towards using it effectively. Most researchers have a general idea about what context is and use that general idea to guide their use of it. However, a vague notion of context is not sufficient; in order to use context effectively, we must attain a better understanding of what context is. A better understanding of context will enable application designers to choose what context to use in their applications and provide insights into the types of data that need to be supported and the abstractions and mechanisms required to support context-aware computing. Previous definitions of context have either been extensional, that is, an enumeration of examples of context, or vague references to synonyms for context.

1.1.1 Previous Definitions of Context

In the work that first introduces the term 'context-aware,' Schilit and Theimer (Schilit and Theimer 1994) refer to context as location, identities of nearby people and objects, and changes to those objects. In a similar definition, Brown *et al.* (Brown, Bovey *et al.* 1997) define context as location, identities of the people around the user, the time of day, season, temperature, *etc*. Ryan *et al.* (Ryan, Pascoe *et al.* 1998) define context as the user's location, environment, identity and time. In previous work (Dey 1998), we enumerated context as the user's emotional state, focus of attention, location and orientation, date and time, objects, and people in the user's environment. These definitions define context by example and are difficult to apply. When we want to determine whether a type of information not listed in the definition is context or not, it is not clear how we can use the definition to solve the dilemma.

Other definitions have simply provided synonyms for context, referring, for example, to context as the environment or situation. Some consider context to be the user's environment, while others consider it to be the application's environment. Brown (Brown 1996b) defined context to be the elements of the user's environment that the user's computer knows about. Franklin and Flaschbart (Franklin and Flaschbart 1998) see it as the situation of the user. Ward *et al.* (Ward, Jones *et al.* 1997) view context as the state of the application's surroundings and Rodden *et al.* (Rodden, Cheverst *et al.* 1998) define it to be the application's setting. Hull *et al.* (Hull, Neaves *et al.* 1997) included the entire environment by defining context to be aspects of the current situation. As with the definitions by example, definitions that simply use synonyms for context are extremely difficult to apply in practice.

The definitions by Schilit *et al.* (Schilit, Adams *et al.* 1994), Dey *et al.* (in our previous work) (Dey, Abowd *et al.* 1998) and Pascoe (Pascoe 1998) are closest in spirit to the operational definition we desire. Schilit, Adams *et al.* claim that the important aspects of context are: where you are, whom you are with, and what

resources are nearby. They define context to be the constantly changing execution environment. They include the following pieces of the environment:

- *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, Abowd *et al.* define context to be the user's physical, social, emotional or informational state. Finally, Pascoe defines context to be the subset of physical and conceptual states of interest to a particular entity. These definitions are too specific. Context is all about the whole situation relevant to an application and its set of users. We cannot enumerate which aspects of all situations are important, as this will change from situation to situation. For example, in some cases, the physical environment may be important, while in others it may be completely immaterial. For this reason, we could not use the definitions provided by Schilit, Adams *et al.*, Dey, Abowd *et al.*, or Pascoe.

1.1.2 Our Definition of Context

Following is our definition of context.

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.*

Context-aware applications look at the *who*'s, *where*'s, *when*'s and *what*'s (that is, what the activities are occurring) of entities and use this information to determine *why* a situation is occurring. An application does not actually determine why a situation is occurring, but the designer of the application does. The designer uses incoming context to determine why a situation is occurring and uses this to encode some action in the application. For example, in a context-aware tour guide, a user carrying a handheld computer approaches some interesting site resulting in information relevant to the site being displayed on the computer (Abowd, Atkeson *et al.* 1997). In this situation, the designer has encoded the understanding that when a user approaches a particular site (the 'incoming context'), it means that the user is interested in the site (the 'why') and the application should display some relevant information (the 'action').

Our definition of context includes not only implicit input but also explicit input. For example, the identity of a user can be sensed implicitly through face recognition or can be explicitly determined when a user is asked to type in her name using a keyboard. From the application's perspective, both are information about the user's identity and allow it to perform some added functionality. Context-awareness uses a generalized model of input, including implicit and explicit input, allowing *any* application to be considered more or less context-aware insofar as it reacts to input. However, in this thesis, we will concentrate on the gathering and use of implicit input by applications. The conceptual framework we will present can be used for both explicit and implicit input, but focuses on supporting the ease of incorporating implicit input into applications.

There are certain types of context that are, in practice, more important than others. These are ***location***, ***identity***, ***time*** and ***activity***. Location, identity, time, and activity are important context types for characterizing the situation of a particular entity. These context types not only answer the questions of who, what, when, and where, but also act as indices into other sources of contextual information. For example, given a person's identity, we can acquire many pieces of related information such as phone numbers, addresses, email addresses, a birth date, list of friends, relationships to other people in the environment, *etc*. With an entity's location, we can determine what other objects or people are near the entity and what activity is occurring near the entity.

This first attempt at a categorization of context is clearly incomplete. For example, it does not include hierarchical or containment information. An example of this for location is a point in a room. That point can be defined in terms of coordinates within the room, by the room itself, the floor of the building the room is in, the building, the city, *etc*. (Schilit and Theimer 1994). It is not clear how our categorization helps to support this notion of hierarchical knowledge. While this thesis will not solve the problem of context categorization, it will address the problem of representing context given that a categorization exists.

1.2 What is Context-Awareness?

Context-aware computing was first discussed by Schilit and Theimer (Schilit and Theimer 1994) in 1994 to be software that "adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time." However, it is commonly agreed that the first research investigation of context-aware computing was the Olivetti Active Badge (Want, Hopper *et al.* 1992) work in 1992. Since then, there have been numerous attempts to define context-aware computing, and these all inform our own definition.

1.2.1 Previous Definitions of Context-Aware

The first definition of context-aware applications given by Schilit and Theimer (Schilit and Theimer 1994) restricted the definition from applications that are simply informed about context to applications that *adapt* themselves to context. Context-aware has become somewhat synonymous with other terms: adaptive (Brown 1996a), reactive (Cooperstock, Tanikoshi *et al.* 1995), responsive (Elrod, Hall *et al.* 1993), situated (Hull, Neaves *et al.* 1997), context-sensitive (Rekimoto, Ayatsuka *et al.* 1998) and environment-directed (Fickas, Kortuem *et al.* 1997). Previous definitions of context-aware computing fall into two categories: using context and adapting to context.

We will first discuss the more general case of using context. Hull *et al*. (Hull, Neaves *et al.* 1997) and Pascoe *et al*. (Pascoe 1998; Pascoe, Ryan *et al.* 1998; Ryan, Pascoe *et al.* 1998) define context-aware computing to be the ability of computing devices to detect and sense, interpret and respond to aspects of a user's local environment and the computing devices themselves. In previous work, we have defined context-awareness to be the use of context to automate a software system, to modify an interface, and to provide maximum flexibility of a computational service (Dey 1998; Dey, Abowd *et al.* 1998; Salber, Dey *et al.* 1999b).

The following definitions are in the more specific "adapting to context" category. Many researchers (Schilit, Adams *et al.* 1994; Brown, Bovey *et al.* 1997; Dey and Abowd 1997; Ward, Jones *et al.* 1997; Abowd, Dey *et al.* 1998; Davies, Mitchell *et al.* 1998; Kortuem, Segall *et al.* 1998) define context-aware applications to be applications that dynamically change or adapt their behavior based on the context of the application and the user. More specifically, Ryan (Ryan 1997) defines them to be applications that monitor input from environmental sensors and allow users to select from a range of physical and logical contexts according to their current interests or activities. This definition is more restrictive than the previous one by identifying the method in which applications acts upon context. Brown (Brown 1998) defines context-aware applications as applications that automatically provide information and/or take actions according to the user's present context as detected by sensors. He also takes a narrow view of context-aware computing by stating that these actions can take the form of presenting information to the user, executing a program according to context, or configuring a graphical layout according to context. Finally, Fickas *et al.* (Fickas, Kortuem *et al.* 1997) define environment-directed (practical synonym for context-aware) applications to be applications that monitor changes in the environment and adapt their operation according to predefined or user-defined guidelines.

1.2.2 Our Definition of Context-Aware

We have identified a novel classification for the different ways in which context is used, that is, the different context-aware features. Following is our definition of context-awareness.

A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.

We have chosen a more general definition of context-aware computing. The definitions in the more specific "adapting to context" category require that an application's behavior be modified for it to be considered context-aware. When we try to apply these definitions to established context-aware applications, we find that they do not fit. For example, an application that simply displays the context of the user's environment to the user is not modifying its behavior, but it is context-aware. If we use the less general definitions, these applications would not be classified as context-aware. We, therefore, chose a more general and inclusive definition that does not exclude existing context-aware applications and is not limited to the other general definitions given above.

1.2.3 Categorization of Features for Context-Aware Applications

In a further attempt to help define the field of context-aware computing, we will present a categorization of features for context-aware applications. There have been two attempts to develop such a taxonomy. The first was provided by Schilit *et al.* (Schilit, Adams *et al.* 1994) and had two orthogonal dimensions: whether the task is to get information or to execute a command and whether the task is executed manually or automatically. Applications that retrieve information for the user manually based on available context are classified as *proximate selection* applications. Proximate selection is an interaction technique where a list of objects (printers) or places (offices) is presented and where items relevant to the user's context are emphasized or made easier to choose. Applications that retrieve information for the user automatically based on available context are classified as *automatic contextual reconfiguration*. It is a system-level technique that creates an automatic binding to an available resource based on current context. Applications that execute commands for the user manually based on available context are classified as *contextual command* applications. They are executable services made available due to the user's context or whose execution is modified based on the user's context. Finally, applications that execute commands for the user automatically based on available context use *context-triggered actions*. They are services that are executed automatically when the right combination of context exists, and are based on simple if-then rules.

More recently, Pascoe (Pascoe 1998) proposed a taxonomy of context-aware features. There is considerable overlap between the two taxonomies but some crucial differences as well. Pascoe's taxonomy was aimed at identifying the core features of context-awareness, as opposed to the previous taxonomy, which identified classes of context-aware applications. In reality, the following features of context-awareness map well to the classes of applications in the Schilit taxonomy. The first feature is *contextual sensing* and is the ability to detect contextual information and present it to the user, augmenting the user's sensory system. This is similar to *proximate selection*, except in this case, the user does not necessarily need to select one of the context items for more information (*i.e.* the context may be the information required). The next feature is *contextual adaptation* and is the ability to execute or modify a service automatically based on the current context. This maps directly to Schilit's *context-triggered actions*. The third feature, *contextual resource discovery*, allows context-aware applications to locate and exploit resources and services that are relevant to the user's context. This maps directly to *automatic contextual reconfiguration*. The final feature, *contextual augmentation*, is the ability to associate digital data with the user's context. A user can view the data when he is in that associated context. For example, a user can create a virtual note providing details about a broken television and attach the note to the television. When another user is close to the television or attempts to use it, he will see the virtual note left previously.

Pascoe and Schilit both list the ability to exploit resources relevant to the user's context, the ability to execute a command automatically based on the user's context and the ability to display relevant information to the user. Pascoe goes further in terms of displaying relevant information to the user by including the display of context, and not just information that requires further selection (*e.g.* showing the user's location vs. showing a list of printers and allowing the user to choose one). Pascoe's taxonomy has a

category not found in Schilit's taxonomy: *contextual augmentation*, or the ability to associate digital data with the user's context. Finally, Pascoe's taxonomy does not support the presentation of commands relevant to a user's context. This presentation is called *contextual commands* in Schilit's taxonomy.

Our proposed categorization combines the ideas from these two taxonomies and takes into account the three major differences. Similar to Pascoe's taxonomy, it is a list of the context-aware features that context-aware applications may support. There are three categories:

1. *presentation* of information and services to a user;
2. automatic *execution* of a service; and,
3. *tagging* of context to information for later retrieval

Presentation is a combination of Schilit's proximate selection and contextual commands. To this, we have added Pascoe's notion of presenting context (as a form of information) to the user. An example of the first feature is a mobile computer that dynamically updates a list of closest printers as its user moves through a building. Automatic execution is the same as Schilit's context-triggered actions and Pascoe's contextual adaptation. An example of the second feature is when the user prints a document and it is printed on the closest printer to the user. Tagging is the same as Pascoe's contextual augmentation. An example of the third feature is when an application records the names of the documents that the user printed, the times when they were printed and the printer used in each case. The user can than retrieve this information later to help him determine where the printouts are that he forgot to pick up.

We introduce two important distinguishing characteristics: the decision not to differentiate between information and services, and the removal of the exploitation of local resources as a feature. We do not to use Schilit's dimension of information vs. services to distinguish between our categories. In most cases, it is too difficult to distinguish between a presentation of information and a presentation of services. For example, Schilit writes that a list of printers ordered by proximity to the user is an example of providing information to the user. But, whether that list is a list of information or a list of services depends on how the user actually uses that information. For example, if the user just looks at the list of printers to become familiar with the names of the printers nearby, she is using the list as information. However, if the user chooses a printer from that list to print to, she is using the list as a set of services. Rather than try to assume the user's state of mind, we chose to treat information and services in a similar fashion.

We chose not to use the exploitation of local resources, or resource discovery, as a context-aware feature. This feature is called automatic contextual reconfiguration in Schilit's taxonomy and contextual resource discovery in Pascoe's taxonomy. We do not see this as a separate feature category, but rather as part of our first two categories. Resource discovery is the ability to locate new services according to the user's context. This ability is really no different than choosing services based on context. We can illustrate our point by reusing the list of printers example. When a user enters an office, their location changes and the list of nearby printers changes. The list changes by having printers added, removed, or being reordered (by proximity, for example). Is this an instance of resource exploitation or simply a presentation of information and services? Rather than giving resource discovery its own category, we split it into two of our existing categories: presenting information and services to a user and automatically executing a service. When an application presents information to a user, it falls into the first category, and when it automatically executes a service for the user, it falls into the second category.

Our definition of context-aware has provided us with a way to conclude whether an application is context-aware or not. This has been useful in determining what types of applications we want to support. Our categorization of context-aware features provides us with two main benefits. The first is that it further specifies the types of applications that we must provide support for. The second benefit is that it shows us the types of features that we should be thinking about when building our own context-aware applications.

1.3 Why is Context Difficult to Use?

We applied our categories of context and context-aware features to a number of well-known context-aware applications. As we will show in the related work section (CHAPTER 2), there is not much of a range in terms of the types of context used and the context-aware features supported in individual applications. Applications have primarily focused on identity and location and generally only present context information to users.

The main reason why applications have not covered the range of context types and context-aware features is that context is difficult to use. Context has the following properties that lead to the difficulty in use:

1. Context is acquired from non-traditional devices (*i.e.* not mice and keyboards), with which we have limited experience. Mobile devices, for instance, may acquire location information from outdoor global positioning system (GPS) receivers or indoor positioning systems. Tracking the location of people or detecting their presence may require Active Badge devices (Want, Hopper *et al.* 1992), floor-embedded presence sensors (Orr 2000) or video image processing.
2. Context must be abstracted to make sense to the application. GPS receivers for instance provide geographical coordinates. But tour guide applications would make better use of higher-level information such as street or building names. Similarly, Active Badges provide IDs, which must be abstracted into user names and locations.
3. Context may be acquired from multiple distributed and heterogeneous sources. Tracking the location of users in an office requires gathering information from multiple sensors throughout the office. Furthermore, context-sensing technologies such as video image processing may introduce uncertainty: they usually provide a ranked list of candidate results. Detecting the presence of people in a room reliably may require combining the results of several techniques such as image processing, audio processing, floor-embedded pressure sensors, *etc*.
4. Context is dynamic. Changes in the environment must be detected in real time and applications must adapt to constant changes. For example, a mobile tour guide must update its display as the user moves from location to location. Also, context information history is valuable, as shown by context-based retrieval applications (Lamming and Flynn 1994; Pascoe 1998). A dynamic and historical model is needed for applications to fully exploit the richness of context information.

Despite these difficulties, researchers have been able to build context-aware applications. But, the applications are typically built using an *ad hoc* process, making it hard to both build new applications and to evolve existing applications (*i.e.* changing the use of sensors and changing the supported application features). Using an *ad hoc* process limits the amount of reuse across applications, requiring common functionality to be rebuilt for every application. Therefore, the goal of this thesis is to support reuse and make it easier to build and evolve applications. The hypothesis of this thesis is:

> *By identifying, implementing and supporting the right abstractions and services for handling context, we can construct a framework that makes it easier to design, build and evolve context-aware applications.*

Through a detailed study of context-aware computing and from our experience in building context-aware applications, we will identify a design process for building context-aware applications. We will examine the design process and determine which steps are common across applications. These steps can be minimized through the use of supporting abstractions that will automatically provide the common functionality and mechanisms that will facilitate the use of these abstractions. The minimized design process is as follows:

1. *Specification*: Specify the problem being addressed and a high-level solution.
2. *Acquisition*: Determine what hardware or sensors are available to provide that context and install them.
3. *Action*: Choose and perform context-aware behavior.

The abstractions and facilitating framework will comprise a toolkit that supports the building and evolution of context-aware applications.

On the building side, designers will be able to easily build new applications that use context, including complex context-aware applications that are currently seen as difficult to build. On the evolution side, designers will easily be able to add the use of context to existing applications, to change the context that applications use and to build applications that can transparently adapt to changes in the sensors they use. By reducing the effort required to build applications, the toolkit will not only allow designers to build more sophisticated applications, but also will also allow them to investigate more difficult issues in context-aware computing such as dealing with inaccurate context and controlling access to personal context. These issues have not previously arisen because the effort required to build applications was so high that it did not allow for any further exploration beyond research prototypes.

1.4 Thesis Contributions

The expected contributions of this thesis are:

- identification of a novel design process for building context-aware applications;
- identification of requirements to support the building and evolution of context-aware applications, resulting in a conceptual framework that both "lowers the floor" (*i.e.* makes it easier for designers to build applications) and "raises the ceiling" (*i.e.* increases the ability of designers to build more sophisticated applications) in terms of providing this support;
- two programming abstractions to facilitate the design of context-aware applications: the context component abstraction (including widgets, aggregators, interpreters and services) and the situation abstraction;
- building of a variety of applications that cover a large portion of the design space for context-aware computing; and,
- building of the Context Toolkit that supports the above requirements, design process and programming abstractions, allowing us and others to use it as a research testbed to investigate new problems in context-aware computing such as the situation programming abstraction and dealing with ambiguous context and controlling access to context.

The five expected contributions of this research are listed above. The first two contributions are intellectual ones. By providing a design process for building context-aware applications, our research will give application designers a better understanding of context and a novel methodology for using context. The identification of the minimal set of requirements for context-aware frameworks will inform other framework or architecture builders in building their own solutions.

The third contribution of our research is to "lower the threshold" (Myers, Hudson *et al.* 2000) for application designers trying to build context-aware applications, through the use of high-level programming abstractions. The goal is to provide an architectural framework that will allow application designers to rapidly prototype context-aware applications. This framework is the supporting implementation that allows our design process to succeed. The functionality and support requirements that will be implemented in our architecture handles the common, time-consuming and mundane low-level details in context-aware computing, allowing application designers to concentrate on the more interesting high-level details involved with actually acquiring and acting on context. The programming abstractions built on top of the architecture will allow designers to think about building applications at a higher level than previously available.

The fourth contribution of our research is an exploration of the design space of context-aware computing. The goal is to build a range of applications that use a large variety of context types (including the four important types identified in 1.1.2) and that use all the context-aware features identified in 1.2.3. By exploring the design space, we can better define it and find the gaps to fuel future research and development.

The fifth contribution of our research is to "raise the ceiling" (Myers, Hudson *et al.* 2000) in terms of what researchers can accomplish in context-aware computing. Our implementation of the architectural framework that we refer to as the Context Toolkit can be used and has been used as a research testbed that allows researchers to more easily investigate problems that were seen as difficult before. These problems include both architectural issues and application issues. For example, on the architecture side, an interesting issue that can be pursued is the use of uncertain context information and how to deal with it in a generic fashion. The architecture with its required set of supporting mechanisms will provide the necessary building blocks to allow others to implement a number of higher-level features for dealing with context. On the application side, the context architecture will allow designers to build new types of applications that were previously seen as difficult to build. This includes context-aware applications that scale along several dimensions, such as number of locations, number of people, and level of availability, with simultaneous and independent activity.

1.5 Thesis Outline

Our definition of context includes not only implicit input but also explicit input. For example, the identity of a user can be sensed implicitly through face recognition or can be explicitly determined when a user is asked to type in their name using a keyboard. From the application's perspective, both are information about the user's identity and allow it to perform some added functionality. Context-awareness uses a generalized model of input, including implicit and explicit input, allowing *any* application to be considered more or less context-aware insofar as it reacts to input. However, in this thesis, we will concentrate on the gathering and use of implicit input by applications.

CHAPTER 2 reviews the related research for this work. This includes an in-depth discussion on existing context-aware applications and demonstrates why existing support for building applications is not sufficient.

CHAPTER 3 introduces the requirements for a conceptual framework that supports the building and evolution of context-aware applications. It also presents the current design process for building applications and shows how the architecture can be used to simplify it.

CHAPTER 4 presents the Context Toolkit, an implementation of the conceptual framework described in CHAPTER 3. The toolkit not only contains this implementation, but also includes a library of reusable components for dealing with context. This chapter also introduces the context component programming abstraction that facilitates the building of context-aware applications.

CHAPTER 5 describes four applications that have been built with the Context Toolkit. Both the applications and their designs are discussed with respect to the Context Toolkit components.

CHAPTER 6 describes our use of the Context Toolkit as a research testbed and our extensions to it. This includes the situation programming abstraction, which further simplifies the process of building and evolving applications. This chapter also includes two explorations into issues usually ignored in context-aware computing, controlling access to context and dealing with ambiguity in context data, and describes how the Context Toolkit facilitated these explorations.

Finally, CHAPTER 7 contains a summary and conclusion with suggestions for future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

The design of a new framework that supports the building and evolution of context-aware applications must naturally leverage off of the work that preceded it. The purpose of this chapter is to describe previous research in the field of context-aware computing. This chapter will focus on relevant context-aware applications, their use of context and their ability to support reuse of sensing technologies in new applications and evolution to use new sensors (and context) in new ways. In CHAPTER 3, after we have introduced the required features of an infrastructure that supports the building of context-aware applications, we will discuss existing infrastructures.

2.1 Context Use

As we stated in the previous chapter, we applied our categories of context and context-aware features to a number of context-aware applications (including those in the references from the previous chapter and the applications discussed later in this chapter). The results are in Table 1 below. Under the context type heading, we present Activity, Identity, Location, and Time. Under the context-aware heading, we present our 3 context-aware features, Presentation, automatic Execution, and Tagging.

There is little range in the types of context used and features supported in each application. Identity and location are primarily used, with few applications using activity (almost half the listed applications that use activity simply used tilting and touching of a device) and time. In addition, applications mostly support only the simplest context-aware feature, that of presenting context information to users. This is partial evidence that there is a lack of support for acquiring a wide range of context from a wide variety of sensors and using the context in a number of different ways.

Table 1: Application of context and context-aware categories

System Name	System Description	Context Type				Context-Aware		
		A	**I**	**L**	**T**	**P**	**E**	**T**
Classroom 2000 (Abowd, Dey *et al.* 1998)	Capture of a classroom lecture			X	X			X
Cyberguide (Long, Kooper *et al.* 1996; Abowd, Atkeson *et al.* 1997)	Tour guide		X	X		X		
Teleport (Brown 1996a)	Teleporting	X	X	X			X	
Stick-e Documents (Brown 1996b; Brown, Bovey *et al.* 1997; Brown 1998)	Tour guide Paging and reminders	 X	X X	X	X	X X		X X
Reactive Room (Cooperstock, Tanikoshi *et al.* 1995)	Intelligent control of audiovisuals	X	X	X			X	
GUIDE (Davies, Mitchell *et al.* 1998)	Tour guide			X		X		
CyberDesk (Dey and Abowd 1997; Dey 1998; Dey, Abowd *et al.* 1998)	Automatic integration of user services	X				X	X	
Responsive Office (Elrod, Hall *et al.* 1993)	Office environment control			X	X		X	
NETMAN (Fickas, Kortuem *et al.* 1997; Kortuem, Segall *et al.* 1998)	Network maintenance			X		X		
Fieldwork (Pascoe 1998; Pascoe, Ryan *et al.* 1998; Ryan, Pascoe *et al.* 1998)	Fieldwork data collection			X	X	X		X
Augment-able Reality (Rekimoto, Ayatsuka *et al.* 1998)	Virtual post-it notes			X		X		X
Active Badge (Want, Hopper *et al.* 1992)	Call forwarding		X	X		X	X	
Manipulative User Interfaces (Harrison, Fishkin *et al.* 1998)	Augmented PDA	X					X	
Tilting Interfaces (Rekimoto 1996)	Augmented PDA	X					X	
Sensing on Mobile Devices (Hinckley, Pierce *et al.* 2000)	Augmented PDA	X					X	
AROMA (Pederson and Sokoler 1997)	Remote awareness of colleagues		X	X		X		
Limbo (Davies, Wade *et al.* 1997)	Communication between mobile workers			X			X	
Audio Aura (Mynatt, Back *et al.* 1998)	Awareness of messages and users		X	X	X	X		

2.2 Methods for Developing Applications

In the previous section, we showed the limited range of context used by context-aware applications and the limited ways in which they used context. In this section, we examine these applications further to elicit the kinds of support necessary for context-aware applications. In particular, we look at applications that suffer from tight coupling between the application and the sensors used to acquire context, applications that support some separation of concerns but that make it difficult to evolve to use new sensors and be notified about changes in context, and applications that are limited in their ability to deal with context.

2.2.1 Tight Coupling

In this section, we will provide examples of applications that have extremely tight coupling to the sensors that provide context. In these examples, the sensors used to detect context were 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. This technique does not support good software engineering practices, by not enforcing separation of concerns between application semantics and the low-level details of context acquisition from individual sensors. The second problem is that there is a loss of generality, making the sensors difficult to reuse in other applications and difficult to use simultaneously in multiple applications. In addition, it is difficult to evolve applications to use different sets of sensors in support of new context-aware features.

2.2.1.1 Manipulative User Interfaces

In the manipulative user interfaces work (Harrison, Fishkin *et al.* 1998), handheld computing devices were made to react to real-world physical manipulations. For example, to flip between cards in a virtual Rolodex, a user tilted the handheld device toward or away from himself. This is similar to the real world action of turning the knobs on a Rolodex. To turn the page in a virtual book, the user "flicked" the upper right or left of the computing device. This is similar to the real world action of grabbing the top of a page and turning it. A final example is a virtual notebook that justified its displayed text to the left or the right, depending on the hand used to grasp it. This was done so the grasping hand would not obscure any text. While this has no direct real world counterpart, it is a good example of how context can be used to augment or enhance activities. Here, sensors were connected to the handheld device via the serial port. The application developers had to write code for each sensor to read data from the serial port and parse the protocol used by each sensor. The context acquisition was performed directly by the application, with minimal separation from the application semantics.

2.2.1.2 Tilting Interfaces

In the similar tilting interfaces work (Rekimoto 1996), the tilt of a handheld computing device was used to control the display of a menu or a map. Here, the sensors were connected via a serial port to a second, more powerful, desktop machine, which was responsible for generating the resulting image to display. The image was sent to the handheld device for display. The entire application essentially resided on the desktop machine with no separation of application semantics and context acquisition. One interesting aspect of this application is that the sensors provided tilt information in a different coordinate system than the application required. The application was therefore required to perform the necessary transformation before it could act on the context.

2.2.1.3 Sensing on Mobile Devices

Hinckley *et al* added tilt sensors, touch sensors and proximity range sensors to a handheld personal computer and empirically demonstrated that the use of context acquired from these sensors provided many benefits (Hinckley, Pierce *et al.* 2000). The sensor information was read in over the computer's serial port for use by a number of applications (although context could only be delivered to the application with the current user focus). Unlike the two previous systems, these applications used the context of multiple

sensors to perform an action. When a user gripped the device like a cell phone close to her ear and mouth and tilted it towards herself, the voice memo application was brought to the forefront. When a user simply changed the orientation of the device, the display changed from using landscape to portrait mode or *vice versa* and scrolled the display either up/down or left/right. Like the two previous research efforts, the sensors were difficult to reuse in multiple applications and could not be used simultaneously by multiple applications. Finally, with the tight coupling between sensors and the applications, the applications would be difficult to evolve to use new sensors or to use the existing sensors to support new application features.

2.2.1.4 Cyberguide

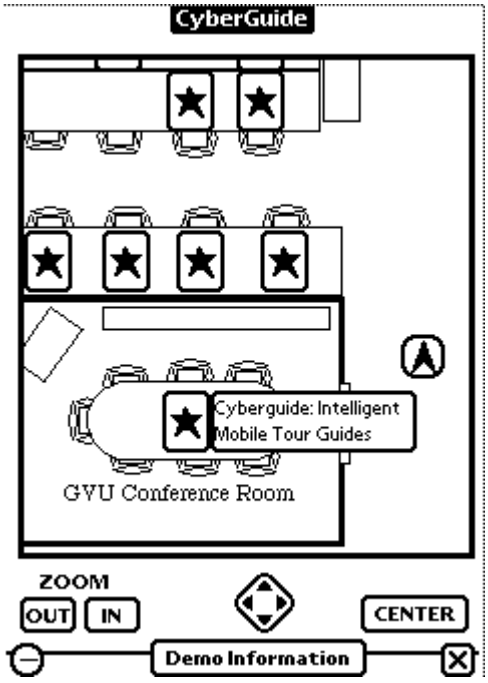


Figure 1: Screenshot of the Cyberguide interface.

The Cyberguide system provided a context-aware tour guide to visitors to a "Demo Day" at a research laboratory (Long, Kooper *et al.* 1996; Abowd, Atkeson *et al.* 1997). The tour guide is the most commonly developed context-aware application (Bederson 1995; Feiner, MacIntyre *et al.* 1997; Davies, Mitchell *et al.* 1998; Fels, Sumi *et al.* 1998; Yang, Yang *et al.* 1999). Visitors were given handheld computing devices. The device displayed a map of the laboratory, highlighting interesting sites to visit and making available more information on those sites (Figure 1). As a visitor moved throughout the laboratory, the display recentered itself on the new location and provided information on the current site. The Cyberguide system suffered from the use of a hardwired infrared positioning system, where remote controls were hung from the ceiling, each with a different button taped down to provide a unique infrared signature (Figure 2). In fact, in the original system, the sensors used to provide positioning information were also used to provide communications ability. This tight coupling of the application and the location information made it difficult to make changes to the application. In particular, when the sensors were changed, it required almost a complete rewrite of the application. As well, due to the static mapping used to map infrared sensors to demonstrations, when a demonstration changed location, the application had to be reloaded with this new information. The use of static configurations had a detrimental impact on evolution of the application.

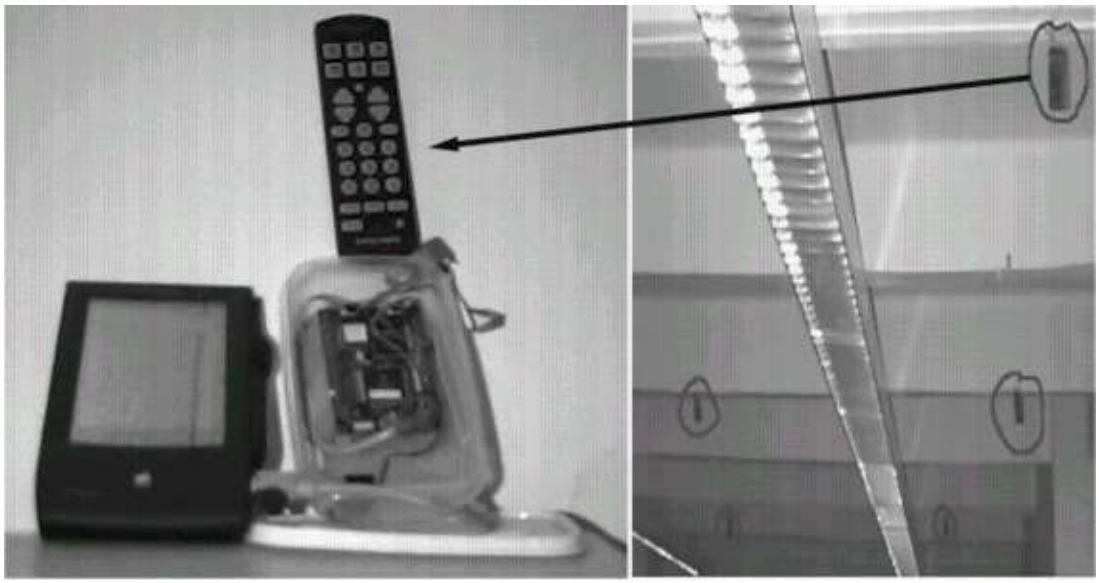


Figure 2: Pictures of the Cyberguide prototype and the infrared-based positioning system.

2.2.2 Use of Sensor Abstractions

In this section, we will discuss systems that have used a sensor abstraction called a server or daemon to separate the details of dealing with the sensor from the application. The sensor abstraction eases the development of context-aware applications by allowing applications to deal with the context they are interested in, and not the sensor specific-details. However, these systems suffer from two additional problems. First, they provide no support for notification of context changes. Therefore, applications that use these systems must be proactive, requesting context information when needed via a querying mechanism. The onus is on the application to determine when there are changes to the context and when those changes are interesting. The second problem is that these servers or daemons are developed independently, for each sensor or sensor type. Each server maintains a different interface for an application to interact with. This requires the application to deal with each server in a different way, much like dealing with different sensors. This still impacts an application's ability to separate application semantics from context acquisition. In addition, the programmer who must implement the abstraction is given no or little support for creating the abstraction. This results in limited use of sensors, context types, and the inability to evolve applications. However, it does solve the problem of inadequate sensor reuse that resulted from tight coupling.

2.2.2.1 Active Badge

The original Active Badge call-forwarding application is perhaps the first application to be described as being context-aware (Want, Hopper *et al.* 1992). In this application, users wore Active Badges, infrared transmitters that transmitted a unique identity code. As users moved throughout their building, a database was being dynamically updated with information about each user's current location, the nearest phone extension, and the likelihood of finding someone at that location (based on age of the available data). When a phone call was received for a particular user, the receptionist used the database to forward the call to the last known location of that user. In this work, a server was designed to poll the Active Badge sensor network distributed throughout the building and to maintain current location information. Servers like this abstract the details of the sensors from the application. Applications that use these servers simply poll the servers for the context information that they collect. This technique addresses both of the problems outlined in the previous section. It relieves application developers from the burden of dealing with the individual sensor details. The use of servers separates the application semantics from the low-level sensor details,

making it easier for application designers to build context-aware applications and allowing multiple applications to use a single server. However, the need for an application to poll a server to determine when an interesting change has occurred is an unnecessary burden, as described earlier. Also, there is no support for creating servers, requiring that each new server (for a new sensor) be written from scratch.

2.2.2.2 Reactive Room

In the Reactive Room project, a room used for video conferencing was made aware of the context of both users and objects in the room for the purpose of relieving the user of the burden of controlling the objects (Cooperstock, Tanikoshi *et al.* 1995). For example, when a figure is placed underneath a document camera, the resulting image is displayed on a local monitor as well as on remote monitors for remote users. Similarly, when a digital whiteboard pen is picked up from its holster, the whiteboard is determined to be in use and its image is displayed both on local and remote monitors. If there are no remote users, then no remote view is generated. Similar to the Active Badge work, a daemon, or server, is used to detect the activity around a specific device. A programmer must individually create a daemon for each device whose activity is being monitored. The daemon abstracts the information it acquires to a usable form for applications. For example, when the document camera daemon determines that a document is placed underneath it, the context information that is made available is whether it has an image to be displayed, rather than providing the unprocessed video signal. While the daemon allows applications to deal with abstracted information rather than device details, the application is required to deal with each daemon in a distinct fashion, impacting the ease of application evolution. In addition, with no support for creating daemons and a new daemon required for each device, it is difficult to use new sensors or devices and context types.

2.2.3 Beyond Sensor Abstractions

In this section, we discuss systems that not only support sensor abstractions, but also support additional mechanisms such as notification about changes in context data, storage of context, or interpretation of context. Interpretation is the transformation of one or more types of context into another type of context. The problem with these systems is that none of them provide all of these features, which are necessary, as we will see in the following chapter.

2.2.3.1 AROMA

The AROMA project attempted to provide peripheral awareness of remote colleagues through the use of abstract information (Pederson and Sokoler 1997). Features were abstracted from audio and video signals captured in colleagues' space. The features were delivered to the other colleagues and rendered in a variety of ways, to investigate whether abstract representations of captured data conveys a sense of remote presence. Its object-oriented architecture used *capture objects* to encapsulate sensors and *abstractor objects* to extract features. 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 with other equivalent ones.

2.2.3.2 Limbo

Limbo is an agent-based system that uses quality of service information to manage communication channels between mobile fieldworkers (Davies, Wade *et al.* 1997). Agents place quality of service information such as bandwidth, connectivity, error rates, and power consumption, as well as location information into tuple spaces. Services that require particular bit rates and connectivity instantiate agents to obtain a satisfactory communications channel. These agents collect quality of service information from the tuple spaces and use this information to choose a communications channel (Friday 1996). Other agents place (and remove) service-related information into (and from) the tuple spaces. These agents and tuple spaces provide an abstraction of the sensor details, not requiring applications (or accessing agents) to deal with the details of the sensor, and allowing them to simply access the sensor data. All the agents have a common interface making it easy for applications to deal with them. This technique allows use of sensor

data by multiple applications and supports distributed sensing and limited interpretation. However, there is no support for notification of changes in context. An agent needs to inspect or query the tuple space to determine whether there is any new relevant context for it to use. In addition, Limbo supports a limited notion of context storage, with only the last context value stored for each context type. This severely limits the ability for applications to act on historical information and trends.

2.2.3.3 NETMAN

The NETMAN system is a collaborative wearable application that supports the maintenance of computer networks in the field (Fickas, Kortuem *et al.* 1997; Kortuem, Segall *et al.* 1998). A field worker uses a wearable computer to diagnose and correct network problems and is assisted by a remote expert. An application on the wearable computer uses the field worker's location, the identities of objects around him and local network traffic information to provide relevant information allowing collaboration between the field worker and the remote network expert. The system uses sensor proxies to abstract the details of the sensors from the application and also supports notification about context updates through a subscription-based mechanism. There is little support for creating new sensor abstractions or proxies, making it difficult to add new sensors. As well, interpretation of sensor data is left up to each application using the context. These issues make it difficult to evolve existing applications and create new applications.

2.2.3.4 Audio Aura

In the Audio Aura system, location, identity and time context were used to control background audio in order to provide serendipitous awareness about physical and virtual information (Mynatt, Back *et al.* 1998). For example, when a user enters a social area, they receive an audio cue via wireless headphones indicating the total number of new email messages they have received and the number from specific people. Also, when a user walks by a colleague's empty office, they hear a cue that indicates how long the colleague has been away for. A server is used that abstracts location and identity context from the underlying sensors (Active Badges and keyboard activity) being used. A goal of this system was to put as much of the system functionality in the server to allow very thin clients. The server supported storage of context information to maintain a history and supported a powerful notification mechanism. The notification mechanism allowed clients to specify the conditions under which they wanted to be notified. However the use of the notification mechanism required knowledge of how the context was actually stored on the server, reducing the separation between context acquisition and context use. In addition, the single server, while providing a single point of contact for the application by aggregating similar types of context together, does not provide support for adding additional sensors and evolving applications to use different sensors.

2.3 Overview of Related Work

In this chapter, we have presented previous applications work that is relevant to providing architectural-level support for building context-aware computing. We presented systems that have extremely tight coupling between the applications and sensors. These systems are difficult to develop due to the requirements of dealing directly with sensors and are hard to evolve because the application semantics are not separated from the sensor details. We presented systems that used sensor abstractions to separate details of the sensors from applications. These systems are difficult to extend to the general problem of context-aware application building because there is no standard abstraction used, with each sensor having its own interface. An application, while not dealing directly with sensor details, must still deal individually with each distinct sensor interface at low levels. In addition, these systems have not supported notification of changes to context data, requiring applications to query and analyze them in order to determine that a relevant change has occurred. Next we presented systems that support additional mechanisms beyond sensor abstraction, including context notification, storage and interpretation. These systems provide only a subset of the required mechanisms for building context-aware applications and do not fully support the ability to reuse sensors or to evolve existing applications to use new sensors and context types.

We have learned quite a bit from these applications. As we will see in the next chapter, these applications support a number of features that would be useful across all context-aware applications. They include

separation of concerns between context acquisition and context use, notification about relevant changes to context values, interpretation of context, storage and aggregation of related context. Our investigation of these applications has informed the design of a conceptual framework that supports context-aware computing.

CHAPTER 3

A CONCEPTUAL FRAMEWORK FOR SUPPORTING CONTEXT-AWARE APPLICATIONS

In CHAPTER 1, we described why context-aware computing is an interesting and relevant field of research in computer science. In CHAPTER 2, we discussed previously built context-aware applications in an attempt to understand what features are common and useful across context-aware applications. In this chapter, we discuss why these applications have been so difficult to build and present a conceptual framework that both provides necessary features and a programming model to make the building process easier. The framework will allow application designers to expend less effort on the details that are common across all context-aware applications and focus their energies on the main goal of these applications, that is, specifying the context their applications need and the context-aware behaviors to implement (Dey, Salber *et al.* 2001).

What has hindered applications from making greater use of context and from being context-aware? As we saw in CHAPTER 2, a major problem has been the lack of uniform support for building and executing these types of applications. Most context-aware applications have been built in an *ad hoc* or per-application manner, heavily influenced by the underlying technology used to acquire the context (Nelson 1998). There is little *separation of concerns*, so application builders are forced to deal with the details of the context acquisition technology. There is little or no support for the variety of features that context-aware applications often require. Finally, there are no programming or building abstractions for application builders to leverage off when designing their context-aware applications. This results in a lack of generality, requiring each new application to be built from the ground up in a manner dictated by the underlying sensing technology.

3.1 Resulting Problems From Context Being Difficult to Use

Pascoe wrote that it is a hard and time-consuming task to create software that can work with variety of hardware to capture context, translate it to a meaningful format, manipulate and compare it usefully, and present it to a user in meaningful way (Pascoe 1998). In general, context is handled in an improvised fashion. Application developers choose whichever technique is easiest to implement, usually dictated by the sensors being used. This comes at the expense of generality and reuse, making it very difficult to integrate existing sensor solutions with existing applications that do not already use context, as well as being difficult to add to existing context-aware applications. As a result of this *ad hoc* implementation, a general trend of tightly connected applications and sensors has emerged that operates against the progression of context-aware computing as a research field and has led to the general lack of context-aware applications. Even when abstractions are used, which are intended to decouple applications and sensors, there is no generalized support for creating these abstractions that satisfy the needs of context-aware computing. This trend has led to three general problems:

- a lack of variety of sensors for a given context type;
- a lack of variety of context types; and,
- an inability to evolve applications.

We will discuss each of these problems and how the *ad hoc* nature of application development has led to them.

3.1.1 Lack of Variety of Sensors Used

In our research in context-aware computing, we have found that there is a lack of variety in the sensors used to acquire context. Sensors not only include hardware devices that can detect context from the physical environment (*e.g.* cameras for face recognition, Active Badges for location), but also software that can detect context from the virtual world (*e.g.* processor load, available screen real estate on a display, last file a user read). The reason for the lack of sensor variety for each type of context is the difficulty in dealing with the sensors themselves. This comes from our own experiences, the experiences of other researchers we have talked with, and the anecdotal evidence from previously developed applications. Sensors are difficult to deal with, as will be shown by the number and difficulty of the sub-steps in Step 2 of the design process given below (Section 3.2).

There is little guidance available to the application designer, other than the requirements of the application. This results in a tight connection between the sensors and the application. The required steps such as determining how to acquire information from a sensor and how to distribute relevant changes to applications are a burden to the application programmer. Basically, these low-level issues are being given equal weight in the design process as the high-level issues such as what context to use and what behaviors to execute. This has resulted in the lack of variety in the types of sensors used for context-aware computing. We see evidence of this when we examine the research performed by various research groups.

Within a single research group, when reuse was planned for, the applications constructed always use the same sensor technology. For example, at Xerox PARC the researchers started with Active Badges, but built their own badge system with greater functionality – the ParcTab (Want, Schilit *et al.* 1995). In all of their context-aware work, they used the ParcTab as the main source of user identity and location (Schilit, Adams *et al.* 1994; Schilit 1995; Mynatt, Back *et al.* 1998). The wearable computing group at Oregon always uses an infrared positioning system to determine location (Bauer, Heiber *et al.* 1998; Kortuem, Segall *et al.* 1998). The Olivetti research group (now known as AT&T Laboratories Cambridge) always uses Active Badges or ultrasonic Active Bats for determining user identity and location (Harter and Hopper 1994; Richardson 1995; Adly, Steggles *et al.* 1997; Harter, Hopper *et al.* 1999). Why is there this reuse of sensors? The answer is twofold. The first part of the answer is that it is convenient to reuse tools that have already been developed. The second part of the answer is that it is often too prohibitive, in terms of time and effort, to use new sensing mechanisms.

This is exactly the behavior we expect and want when we are dealing with a particular context type. If there is support for a particular type of sensor, we expect that application programmers will take advantage of that support. The problem is when an application programmer wants to use a new type of context for which there is no sensor support or when a combination of sensors is needed. The lack of this sensor support usually results in the programmer not using that context type or combination of sensors. The difficulty in dealing with sensors has hurt the field of context-aware computing, limiting the amount and variety of context used. Pascoe echoed this idea when he wrote that the plethora of sensing technologies actually works against context-awareness. The prohibitively large development required for context-aware computing has stifled more widespread adoption and experimentation (Pascoe 1998). We could adapt to this trend of using fewer sensors and investigate whether we can gather sufficient context from a single (or a minimal set) of sensor(s) (Ward 1998). However, this does not seem fruitful – the diversity of context that application designers want to use cannot be captured with a handful of sensors. Instead, we should provide support to allow application designers to make use of new sensors more easily.

3.1.2 Lack of Variety of Types of Context Used

As introduced in the previous section, stemming from the lack of variety in sensors, we have the problem of there being a lack of diversity in the types of context that are used in context-aware applications. The lack of context limits applications by restricting their scope of operation. In general, most context-aware applications use location as their primary source of context (Schmidt, Beigl *et al.* 1998; Dey and Abowd

2000b). Context-aware applications are limited by the context they use. The lack of context types has resulted in the scarcity of novel and interesting applications.

There is an additional problem that arises directly from the use of *ad hoc* design techniques. As stated before, sensors have usually not been developed for reuse. Software is written for sensors on an individual basis, with no common structure between them. When an application designer wants to use these sensors, she finds that the task of integrating the application with these sensors is a heavyweight task, requiring significant effort. This affects an application's ability to use different types of context in combination with each other. This results in fairly simplistic context-aware applications that use only one or a few pieces of context at any one time.

3.1.3 Inability to Evolve Applications

An additional problem that comes from the tight connection of applications to sensors is the static nature of applications. Ironically, applications that are meant to change their behavior when context changes have not shown the ability to adapt to changes in the context acquisition process. This has made applications difficult to evolve on two fronts in particular: movement of sensors and change in sensors or context. When a sensor is moved to a new computing platform, the application can no longer communicate with it unless it is told about the move. In practice, changes made to the application are not something that occurs at runtime. Instead, an application is shut down, the new location information is hardcoded into it, and then it is restarted.

Let us use the example of an In/Out Board that is commonly found in offices (Moran, Saund *et al.* 1999). The board is used to indicate which members of the office are currently in the building and which are not. In both the academic and corporate world, we often find ourselves trying to determine whether someone is in the office in order to interact with her. With traditional In/Out Boards, an occupant of an office enters the office and moves her board marker from the 'out' column to the 'in' column. When she leaves, she moves her marker from the 'in' column back to the 'out' column. An electronic In/Out Board can use a sensor to automatically determine when users arrive at or leave the office. If an In/Out Board were using a face recognition sensor system that was moved to the entrance to the building, from the entrance to the laboratory, the application would have to be stopped, its code modified to use the sensor at its new location, recompiled and then restarted. Instead, an application should automatically adjust to the sensor movement and start using the sensor after it has been moved, with no intervention required. This is a minor annoyance in this situation, but has the potential to be a maintenance nightmare if several applications are deployed and hundreds or even thousands of sensors are moved. One of our goals in this research is to produce a system that can deal with large numbers of applications, services and sensors simultaneously.

When the sensor used to obtain a particular piece of context is replaced by a new sensor or augmented by an additional sensor, the evolution problem is much more difficult. Because of the tight coupling of the application to the sensors, an application often needs a major overhaul, if not a complete redesign, in this situation, as we found when trying to augment Cyberguide (Abowd, Atkeson *et al.* 1997). This also applies when the designer changes or adds to the context being used. We will revisit the In/Out Board application again. If an Active Badge system were used to acquire identity rather than a face recognizer (or was added to be used in concert with the face recognizer), then a large portion of the application may need to be rewritten. The application would have to be modified to use the communications protocol and communications and event mechanisms dictated by the Active Badge system. The portion of the application that performs conversions on the sensed context and determines usefulness will also require modification. The difficulties in adapting applications to changes in how context is acquired results in relatively static applications. This leads to applications that are short-lived in duration, which is opposed to the view in ubiquitous computing where computing services are available all the time (for long-term consecutive use). The inability to easily evolve applications does not aid the progress of context-aware computing as a research field.

3.2 Design Process for Building Context-Aware Applications

To understand more fully the difficulty in building context-aware applications, we need to investigate the design process for building these applications. We have identified a design process for building context-aware applications. We believe that the difficulty in building context-aware applications has been the lack of infrastructure-level support for this design process. The design process (adapted from (Abowd, Dey *et al.* 1998)) is as follows:

1. *Specification*: Specify the problem being addressed and a high-level solution.
   1.1. Specify the context-aware behaviors to implement.
   1.2. Determine what collection of context is required for these behaviors to be executed, using any context-acquisition mechanisms that already exist.
2. *Acquisition*: Determine what new hardware or sensors is needed to provide that context.
   2.1. Install the sensor on the platform it requires.
   2.2. Understand exactly what kind of data the sensor provides.
   2.3. If no application programming interface (API) is available, write software that speaks the protocol used by the sensor.
   2.4. If there is an API, learn to use the API to communicate with the sensor.
   2.5. Determine how to query the sensor and how to be notified when changes occur.
   2.6. Store the context.
   2.7. Interpret the context, if applicable.
3. *Delivery*: Provide methods to support the delivery of context to one or more, possibly remote, applications.
4. *Reception*: Acquire and work with the context.
   4.1. Determine where the relevant sensors are and how to communicate with each
   4.2. Request and receive the context.
   4.3. Convert it to a useable form through interpretation.
   4.4. Analyze the information to determine usefulness.
5. *Action*: If context is useful, perform context-aware behavior.
   5.1. Analyze the context treating it as an independent variable or by combining it with other information collected in the past or present.
   5.2. Choose context-aware behavior to perform.

We will show that for each of these steps, there are general infrastructure-level supporting mechanisms that are required by all but the most trivial context-aware applications. It is important to note that the goal of a context-aware application designer is to provide context-aware services or behaviors that are modified based on some specified context. The designer does not want to worry about the middle three steps, but instead would like to concentrate on the specifying and performing the actual context-aware behaviors. Furthermore, it is these three steps that make building context-aware applications difficult and time-consuming.

3.2.1 Using the Design Process

To illustrate the design process, we will discuss how an In/Out Board application would have been developed without any supporting mechanisms. Here, the In/Out Board application is in a remote location from the sensors being used. In the first step of the design process, *specification*, the developer specifies the context-aware behavior to implement. In this case the behavior being implemented is to display whether occupants of a building are in or out of the building and when they were last seen (step 1.1). Also, the developer determines what context is needed. In this case, the relevant context types are location, identity and time (step 1.2). We will assume that there is no existing support in the building to provide this context, so a new sensor has to be used.

The second step, *acquisition*, is where the developer deals directly with the sensors. Java iButtons® (Dallas Semiconductor 1999) are chosen to provide the location and identity context. An iButton is a

microprocessor that contains a unique identity. This identity can be read when the iButton is docked with a reader. A reader is installed on a computer, at the entrance to the research lab (step 2.1). The reader combined with the iButton provides an abstract identity – rather than providing a name, which is what the developer wants, the iButton provides a 16 character hexadecimal value (step 2.2). The location context is simply the location of the installed reader, and time context is determined from the computer being used. The iButton and reader come with an API, so the developer writes supporting code that reads the abstract identity when an iButton is docked with a reader (step 2.3/2.4). This code must support both querying and notification of events (step 2.5), where, informally, an event is a timely difference that makes a difference (Whitehead, Khare *et al.* 1999). Querying allows an application to determine current status and notification allows an application to maintain its knowledge of status over a period of time. The context information acquired from the sensor must be stored (step 2.6) so additional code is required from the developer. If the application is ever restarted, it must have the ability to query for the current status of all of its users. This information would only be available if the context were stored for later use. The final step in acquisition is to interpret the context, if possible and applicable (step 2.7). While the abstract identity does need interpretation into an actual user name, the interpretation is usually left to the application.

The third step, *delivery*, deals with making context available to a remote application. Here, the developer must design and implement a communications protocol for allowing applications to access the acquired context. The protocol must provide support for both querying and notification schemes and must be able to interact with the code that actually acquires the context. Finally, a communications mechanism must be developed to support the protocol, allowing the application to actually communicate with the sensor (and associated code).

In the fourth step, *reception*, the developer begins to work on the application side of the problem, determining where the sensors are and how to communicate with them, asking the sensor for context and performing some initial analysis. The In/Out Board needs to acquire current state information for who is in and who is out of the building, so it first locates the sensor and then queries it. It wants to maintain accurate state information, so it also subscribes to the iButton asking to be notified of relevant changes (step 4.1). To receive the context, the application must adhere to the same protocol and communications mechanism that the iButton uses (from step 3). The communications is written to specifically deal with the iButton and accompanying code. Once the application receives context information (location of the reader, abstract identity and time of docking), it must convert the abstract identity to a usable form (step 4.2). It can use a simple hash table that has an associated user name for each abstract identity. After this interpretation, it analyzes the information to see if the context is useful. In this case, it means whether the iButton user was a registered user of the In/Out Board (step 4.3). If so, the application can use the context; otherwise, it discards the context and waits for new context to arrive.

In the fifth and final step, *action*, the developer actually uses the context to implement and perform some context-aware behavior. The application must determine if the context it received was for a user arriving or leaving the building. When new useful context arrives for a given user, that user's state is toggled from in to out, or *vice-versa* (step 5.1). Finally, the developer can act on the context, and display the user's new state on the In/Out Board (step 5.2).

3.2.2 Essential and Accidental Activities

Considering this is a fairly straightforward application, this design process is quite long and not easy to follow. There are too many steps in the design process that are redundant or replicated across individual applications. By providing uniform support for these steps, we can reduce the process to a smaller and, therefore, simpler set of tasks. In this section we will examine what steps can be removed or reduced in the design process.

In his famous essay on software engineering practices, Fred Brooks distinguished essential and accidental activities in software development (Brooks 1987). Essential activities are fundamental to constructing a piece of software and include understanding a problem and modeling a solution for that problem. They are

activities specific to the actual application being designed. Accidental tasks are ones that are only necessary because support is not available to handle or take care of them. These are tasks that are common across context-aware applications. If we can reduce the design process for building context-aware applications to a set of essential activities, we can ease the burden of building these types of applications. We will now discuss the design process in terms of Brooks' essential and accidental activities.

3.2.2.1 Specification

In step 1, a context-aware application designer must specify and analyze the context-aware behaviors. From this analysis, the designer will determine and specify the types of context she requires. This is an essential activity, for without it, there can be no use of context. This step is the focal point of modeling and designing context-aware applications, and typically impacts the rest of the application design.

3.2.2.2 Acquisition

In step 2, the application designer determines what sensors, hardware or software, are available to provide the specified context from step one. This step can be essential or accidental, depending on the sensor chosen. If the sensor has been used before, then this step and all of its sub-steps are accidental. These steps should already have been performed and the resulting solution(s) should be reusable by the application designer.

If the sensor has not been used before, then all of the difficult sub-steps must be completed. Installation of a sensor can be particularly troublesome when the platform required by the sensor does not match the platforms available or being used by the application. It could require the use of distributed computation and/or multiple types of computing platforms, both of which cause burdens to application programmers. Acquiring data from a sensor is a time-consuming and difficult step. The designer must understand exactly the type of data provided by the sensor, in order to determine what context information can be derived from it. For example, a GPS receiver and a smart card with accompanying reader may both appear to provide location context. In fact, the GPS receiver provides current and continuous latitude and longitude location information. However, the smart card reader only indicates the presence of an individual at a particular location for a particular instant in time. While both of these can be seen as providing location information, they provide quite distinct information.

If an API is not available for the sensor, the application designer must determine how to communicate with the sensor and acquire the necessary data. If an API is available, the designer usually has an easier job in acquiring the sensor's data. But she still has to ensure that there is functionality available to support both querying of the sensor and notification by the sensor when data changes. After the context information has been acquired, it must be converted into a useful form. This can either happen at the sensor level or at the application level. If done at the sensor level, it relieves the application designer of the burden of implementing it herself, and allows reusability of the functionality by multiple applications. If performed at the application level, the application can specify how the interpretation/conversion should be done. In either case, this step is accidental. If the data were available in the form desired, this issue would not be a concern to the application designer. As previously mentioned, context information also needs to be stored. Storage is an accidental step because it is common across all sensors and is not strictly dependent on the sensor being used. Designers simply want to take advantage of stored context and not worry about how it is supported.

3.2.2.3 Delivery

Once in the correct format, the context information must be delivered to the application (step 3). This requires the specification of a querying and notification protocol and communications protocol. This is more complex if the application is not local to the sensor, because a mechanism that supports distributed communications must also be developed. The delivery of context is an accidental step. The designer merely wants to acquire the context and should not be concerned about these types of low-level details.

3.2.2.4 Reception

In the reception step (4), an application must first receive the context information. To receive the context, it must know the location of the sensor so it can communicate with it. It can communicate using a querying or a subscription mechanism. As explained in the previous sub-section, the application must use the same communication protocols and mechanisms specified by the sensor(s) providing the context. Once received, the context must be converted to a useful format, if applicable. Using the previous example of the GPS receiver, an application may require the name of the street the user is on, however the receiver only provides latitude and longitude information. The application must interpret this latitude and longitude information into street-level information so that it can use the data. Finally, the application must analyze the context to determine if it is useful. For example, an application may only want to be notified when a user is located on a given subset of streets. The sensor, however, may provide all available context, regardless of what the application can use. In this case, the application must analyze the context to determine the usefulness of the received context. The reception step is accidental. Again, the designer most likely does not care about the details of context reception and determining usefulness, and is concerned only with receiving useful context.

3.2.2.5 Action

If the context is useful information, the application designer provides functionality to further analyze the context to determine what action to take (step 5). This ranges from analyzing the types of context received to analyzing the actual values of the context. This analysis is accidental and should be automatically provided by the supporting infrastructure. Finally, the action includes choosing a context-aware behavior to perform as well as indicating how the behavior should be executed. This last action step is application-specific and is essential. It is or should be the main emphasis in building context-aware applications, that of actually acting on the acquired useful context.

3.2.3 Revised Design Process

We have described the design process for context-aware computing. We have shown that when the steps in the process are reduced to what is essential or necessary, we are left with the following simpler and novel design process:

1. *Specification*: Specify the problem being addressed and a high-level solution.
   1.1. Specify the context-aware behaviors to implement.
   1.2. Determine what context is required for these behaviors (with a knowledge of what is available from the environment) and request it.
2. *Acquisition*: Determine what hardware or sensors are available to provide that context and install them.
3. *Action*: Choose and perform context-aware behavior.

Note that if all necessary context acquisition components are already available for use by applications, then step 2 is not required, and we are left with a design process that consists of specifying the problem and performing a context-aware service when the specified context is received.

To help designers use this reduced design process that we have derived, we must provide infrastructure-level support. In addition, we have identified the negative trend of connecting application design too tightly to sensors used that has resulted from the use of the longer, more complex design process. We have also identified three basic problems that have emerged due to this trend: the lack of variety in sensors used, the lack of variety in the types of context used and the inability to evolve context-aware applications. This analysis of the design process and the resulting problems has demonstrated a gap in the research in context-aware computing. This has allowed us to focus our research on infrastructure support and has lead to our thesis statement:

By identifying, implementing and supporting the right abstractions and services for handling context, we can construct a framework that makes it easier to design, build and evolve context-aware applications

3.3 Framework Features

In order to support the reduced design process presented in the previous section, we need to identify the required features of a supporting framework. In CHAPTER 2, we presented previous work in the area of context-aware applications. While no one application used a complete solution that would support our reduced design process, each hinted at least a partial solution. The major goal of this research is to allow context-aware application developers to more easily build applications and to support them in building more complex applications than they have been able to build so far.

The set of required features for a supporting framework will be derived from both the accidental steps of the design process and common features found across multiple context-aware applications. For the sake of simplicity, we will assume for now that the users of this conceptual framework are not going to modify or add to the architecture, and simply use the existing support and architectural components. Hudson refers to this class of users as *library programmers* (Hudson 1997). There is a second class of users, whom Hudson refers to as *toolkit programmers*, which will modify and add components to the architecture because they do not already exist. We will refer to both sets of users as *designers* or *developers*. In an upcoming section (3.5), we will discuss the support that toolkit programmers require.

There are a number of basic framework features required by library programmers:

- Context specification (CS)
- Separation of concerns and context handling (SEP)
- Context interpretation (I)
- Transparent distributed communications (TDC)
- Constant availability of context acquisition (CA)
- Context storage (ST)
- Resource discovery (RD)

By addressing these features in a framework, we will enable library programmers to more easily build context-aware applications. Builders will be able to leverage off of the framework, rather than be forced to provide their own general support in an *ad hoc* manner.

3.3.1 Context Specification

Perhaps the most important requirement of a framework that supports our minimized design process is a mechanism that allows application builders to specify what context an application requires. If we examine the design process, we see that there are only two steps: specifying what context an application needs and deciding what action to take when that context is acquired. The specification mechanism and, in particular, the specification language must allow application builders to indicate their interest along a number of dimensions:

- single piece of context vs. multiple pieces of context. For example, a single piece of context could be the location of a user and multiple pieces of context could be the location of a user and the amount of free time she has before her next appointment;
- if multiple, related context vs. unrelated context. Related context is context dealing with a single entity. For example, related context about a user could include her location and the amount of free time she has, and unrelated context could be the location of a user and the share price of a particular stock;
- unfiltered context vs. filtered context. For example, an unfiltered request to be notified about a user's change in location would result in the requesting entity being notified each time the user

moved to a new location. In contrast, a filtered request might allow the requesting entity to be notified only when the user moved to a different building; and,

- uninterpreted context vs. interpreted context. For example, a GPS returns its location in the form of a latitude and longitude pair. If an application is interested in street-level information and no sensor provides that information but there exists an interpretation capability that converts latitude-longitude pairs to street name, then the request should be satisfied using a combination of the sensor and the interpretation.

This runtime mechanism will ensure that when applications receive the context they have requested, it will be useful to the application and will, ideally, not require further interpretation or analysis. The mechanism should notify the application whether the specified context is available from the infrastructure or not. This notification allows the application to decide whether it needs to change its context specification or to notify the end user. Finally, the mechanism must support multiple specifications from multiple applications that may be requesting the same or different set of context.

This mechanism makes it easier for application designers to specify what context their applications require. It helps remove the accidental design process steps of *reception* (receiving the context, interpretation and analysis) and the first step of *action* (further analysis with additional context to determine usefulness).

3.3.2 Separation of Concerns and Context Handling

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. As we showed earlier, there are two common ways in which context is handled: connecting sensor drivers directly into applications (2.2.1) and using servers to hide sensor details (2.2.2). Both of these methods result in applications that are difficult to build and evolve.

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 computing, but also with pen and speech input (Arons 1991), and with the unconventional input devices used in virtual reality (MacIntyre and 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.

In our review of related work, we discussed applications that supported event management, either through the use of querying mechanisms, notification mechanisms, or both to acquire context from sensors. It is not a requirement that both 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, Cheverst *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.

There have been previous systems that handle context in this way (Schilit 1995; Bauer, Heiber *et al.* 1998). These attempts used servers that support both a querying mechanism and a notification mechanism. The notification mechanism relieves an application from having to poll a server to determine when interesting changes occur. However, this previous work has suffered from the design of specialized servers, which result in the lack of a common interface across servers (Bauer, Heiber *et al.* 1998), forcing applications to deal with each server in a distinct way. This results in a minimal range of server types being used (e.g. only location (Schilit 1995)).

By separating how context is acquired from how it is used, we are able to greatly reduce the design process step of *acquisition*. Applications can now use contextual information without worrying about the details of a sensor and how to acquire context from it.

3.3.3 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. 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 is used to check co-location. At the next level, this information could be combined with sound level information to determine if a meeting is taking place. From an application designer's perspective, the use of these multiple layers should be transparent. In order 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. In order for the interpretation to be easily reusable by multiple applications, it needs to be provided by the framework. Otherwise, each application would have to re-implement the necessary implementation. We developed a mechanism to perform transparent recursive interpretation in our previous work on CyberDesk (Dey, Abowd *et al.* 1998). For example, the CyberDesk infrastructure would automatically convert the selected text "Anind Dey" to a name and then to its corresponding email address "anind@cc.gatech.edu".

The interpretation mechanism removes the interpretation sub-steps found in both the *acquisition* and *reception* steps of the design process. Combined with the specification mechanism, it allows applications to simply request the context they want, regardless of whether interpretation is required or not, and for the applications to receive the requested context.

3.3.4 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 (CHAPTER 1), we see that time is a

very important type of context. In order 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. However, in some cases, just knowing the ordering of events or causality is sufficient.

Transparent communication allows the *delivery* step of the design process to be removed completely along with the first step of the *reception* step, and is an integral part of the context specification mechanism. The provision of a global timeclock makes complex analysis of context possible by allowing context data to be compared on a temporal basis.

3.3.5 Constant Availability of Context Acquisition

With graphical user interfaces (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, as needed.

Because these components run independently of applications, there is a need for them to be persistent, 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, Hopper *et al.* 1992). When a phone call was received, the receptionist forwarded the call to the phone nearest the intended recipient, after determining the recipient's location. The application that displayed user locations 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.

Constant availability of context acquisition components facilitates the ability to locate existing context-sensing components found in the *specification* step. Because these components are persistent and are independent from applications, they can be found and used by any application. If existing components can be found, the need for the *acquisition* step is removed.

3.3.6 Context Storage

A requirement linked to the need for constant availability is the desire to maintain historical information. GUI 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 implement 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, in order to predict his future location. For these reasons, the framework must support the storage of context. Obviously, this feature removes the sub-step dealing with the storing of context in the *acquisition* step of the design process. It also allows complex analysis to be performed by providing a source of historical/past context.

3.3.7 Resource Discovery

In order for an application to communicate with a sensor (or rather its software wrapper), it must know what kind of information the sensor can provide, where it is located and how to communicate with it (protocol 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 framework needs to support a form of resource discovery (Schwartz, Emtage *et al.* 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 In/Out Board application, rather than hardcoding the location of the iButton reader 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 resource discovery feature would not only be useful at runtime, but also at design-time. An application designer could use the resource discovery mechanism to determine whether the context she requires is already available in the environment or in combination with the specification mechanism and the constant availability of context components to determine what situations can be sensed and whether a given context request can be fulfilled by the currently executing infrastructure. If the current infrastructure does not provide the needed context, the designer will know what components need to be added.

A resource discovery feature allows us to remove the first part of the *reception* step in the design process. With this feature, applications do not need to explicitly or *a priori* know where specific sensors are. Instead, in combination with the specification mechanism, they can request a certain type of context and be automatically connected to the sensor or interpreter that can satisfy the request. In addition, at design time, a developer can locate what components already exist in the environment, allowing her to determine what components, if any, need to be added, simplifying the *specification* step.

3.4 Existing Support for the Architectural Features

In the previous section, we introduced a set of seven required features for an architecture that supports the building and execution of context-aware applications. In this section, we discuss existing architectures not intended for context-awareness, existing architectures built to support context-awareness and some proposed architectures that are relevant for context-aware computing. In particular, we examine the support (existing or proposed) for these required features and will show that none of these architectures provide the required support. This provides the initiative for us to present a new conceptual framework that supports the design process and required features introduced in this chapter.

3.4.1 Relevant Non-Context-Aware Architectures

In this section, we discuss three architectures that were not intended to support context-aware applications. They are relevant because they support separation of concerns between the acquisition and use of data (implicit and explicit) and some of the other required features for context-awareness.

3.4.1.1 Open Agent Architecture

The Open Agent Architecture (and the follow-up Adaptive Agent Architecture) is an agent-based system that supports task coordination and execution (Cohen, Cheyer *et al.* 1994; Kumar, Cohen *et al.* 2000). While it has been mostly used for the integration of multimodal input, it is applicable to context-aware computing. In this system, agents represent sensors and services. 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 appears 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. The architecture supports automatic interpretation, querying

and notification of information, and distributed computation. It suffers from being built to only support a single application at a time and not being able to handle runtime changes to the architecture (*e.g.* agents being instantiated or killed during application execution). It was not intended for long-term persistent use, dealing with neither storage nor support for agents running independently of applications, nor was it meant to handle a large number of applications, services and sensors.

3.4.1.2 Hive

Hive is a platform for distributed software agents (Minar, Gray *et al.* 2000). It consists of *cells* that are network accessible pieces of computation that host both *shadows*, encapsulations of local resources (sensors and other devices), and *agents* that use these local resources and communicate with each other in service of some application. Hive has been primarily used to connect devices together in support of some greater application. This includes a jukebox application that plays a particular audio file when the user places a particular object on a table or when a user walks into her office and an instrumented kitchen that helps you follow a recipe by sensing various activities and appliance statuses (weighing ingredients, oven temperature, ingredients currently being used). The shadows provide some separation of concerns although each shadow provides a separate interface, complicating how an application or agent can interact with the shadows. Hive agents run autonomously from individual applications and communicate using Java Remote Method Invocation (Sun Microsystems 2000a), allowing them to support the transparent distributed communications and constant availability requirements. However, there is no explicit support for storage or interpretation of context and limited support for specifying what context or data an application is interested in.

3.4.1.3 MetaGlue

MetaGlue is another software agents platform (Coen, Philips *et al.* 1999; Phillips 1999). Here, agents are used to represent both local resources and interactions with those resources. MetaGlue supports a single user as he explicitly interacts with Hal, an intelligent room. This includes telling the user how many messages he has when he enters the room, asking the user if he's asleep when the user is lying down on the couch for an extended period of time, and changing the room's lighting if the user is asleep. In addition, the user can interact with information displays and request additional information about a portion of the display the user is pointing at. In many ways it is similar to the Hive system just discussed. It provides a custom distributed communications scheme to all agents, so an agent creator does not need to worry about communications. Agents run autonomously from individual applications so they are always available to service multiple applications. In fact, when an existing agent is not currently running but is needed by an application or agent, the application or agent can automatically start the agent. Similar to Hive, there is no explicit support for interpretation of context or storage of context and limited support for specifying what context an agent is interested in.

3.4.2 Context-Aware Architectures

In this section, we discuss architectures that have been specifically developed to support context-aware applications. These architectures were designed to be applicable to a range of context-aware applications, but, in fact, deal with only a portion of the context-aware application feature space.

3.4.2.1 Stick-e Notes

The Stick-e notes system is a general framework for supporting a certain class of context-aware applications (Brown 1996b). Whereas our research is looking at supporting the acquisition and delivery of context, this research focuses on how to support application designers in actually using the context to perform context-aware behaviors. The goal of this work is to allow non-programmers to easily author context-aware services. It provides a general mechanism for indicating what context an application designer wants to use and provides simple semantics for writing rules that specify what actions to take when a particular combination of context is realized. For example, to build a tour guide application with this architecture, an author would write individual rules, in the form of stick-e notes. An example note to

represent the rule "When the user is located between the location coordinates (1,4) and (3,5) and is oriented between 150 and 210 degrees during the month of December, provide information about the cathedral", is given in Figure 3:

```
<note>
      <required>
      <at> (1,4) .. (3,5)
      <facing> 150 .. 210
      <during> December
<body>
      The large floodlit building at the bottom of the hill is
      the cathedral. (Brown, Bovey et al. 1997)
```

Figure 3: Example Stick-e Note.

The term <required> means that the following context fields must be matched by the user's context in order for the text to be displayed. The term <at> refers to a rectangle in system-specific coordinates that the user must be located within. The term <facing> refers to the orientation of the user. The note portion of the stick-e note provides a description of the user's context. The body portion indicates the action that is to be taken when the context is matched. In this case, the textual string is displayed to the user. Each note such as this one represents a rule that is to be fired when the indicated context requirements are met. A group of notes or rules are collected together to form a stick-e document. The application consists of the document and the stick-e note architecture.

The approach, while interesting, appears to be quite limited due to the decision to support non-programmers. The semantics for writing rules is quite limited and cannot handle continuous or even frequently changing discrete data. It is not meant for programmers to integrate into their applications. It provides a central mechanism for determining when the clauses in a given rule have been met, limiting scalability. While no information is provided on how this mechanism acquires its context or on how the centralized mechanism determines that the required context has indeed been collected, the mechanism does hide the details of the acquisition from applications. There is no support for querying for context, storing context, or interpreting context.

3.4.2.2 Sulawesi

Sulawesi is a framework that supports multimodal interaction on a wearable computer (Newman 1999). It is primarily aimed at supporting a user interacting with computational services on her wearable computer using a mixture of speech (recognized and synthesized), visual input and output and keyboard input. However, agents that effectively wrap or encapsulate each of these input and output devices have also been used to encapsulate sensors like GPS units to acquire context information. The wearable computing system monitors the user's environment (primarily location and movement) to determine what the best way to proactively deliver information to the user. For example, if the user is walking, the system may present information (*e.g.* an incoming message or a location-based reminder) using synthesized speech rather than graphically, since a graphical display is harder to concentrate on and read while in motion. The agents used in Sulawesi provide partial separation of concerns, by presenting a representation of the acquired data, but force applications and other agents to know details of how the data was acquired. There is limited support for resource discovery allowing agents to be added and removed at runtime and limited support for context specification and being notified about changes to context information. There is no support for interpretation, distributed communications or storage of context.

3.4.2.3 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 and 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 or storage of context. As well, while 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.

3.4.2.4 CyberDesk

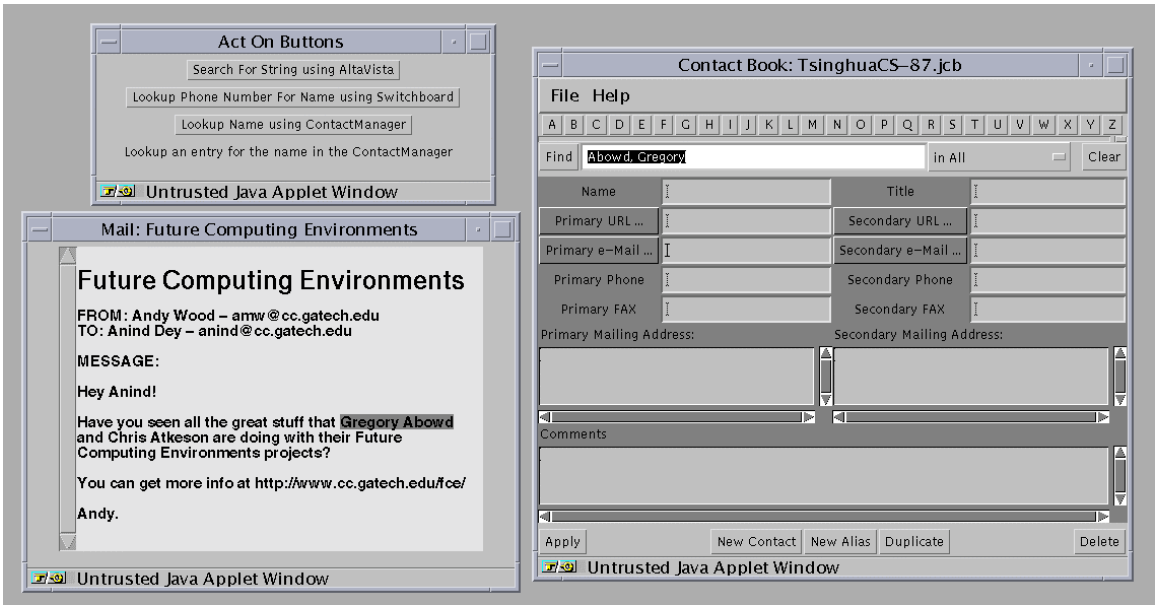


Figure 4: CyberDesk screenshot where services are offered that are relevant to the user's name selection.

In our previous research on context-aware computing, we built an architecture called CyberDesk (Dey and Abowd 1997; Dey 1998; Dey, Abowd *et al.* 1998). This architecture was built to automatically integrate web-based services based on virtual context, or context derived from the electronic world, similar to Apple's Data Detectors (Nardi, Miller *et al.* 1998) and Intel's Selection Recognition Agent (Pandit and Kalbag 1997). The virtual context was the personal information the user was interacting with on-screen including email addresses, mailing addresses, dates, names, URLs, *etc*. An example application is when a user is reading her e-mail and sees that there is an interesting message about some relevant research (Figure 4). She highlights the researcher's name, spurring the CyberDesk architecture into action. The architecture attempts to convert the selected text into useful pieces of information. It is able to see the text as simple text, a person's name, and an email address. It obtains the last piece of information by automatically running a web-based service that convert names to email addresses. With this information, it offers the user a number of services including: searching for the text using a web-based search engine, looking up the name in her contact manager and looking up a relevant phone number using the web.

While it was limited in the types of context it could handle, it 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. For example, in Figure 5, a user is

looking at her schedule for the current day and selects the name of a researcher that she is meeting with. CyberDesk offers to perform the same services as in Figure 4, but also has converted the researcher's name to an e-mail address, a phone number, a physical address (the researcher's home address), and a web page (the researcher's home page) and offers services for each of these. In addition, these converted pieces of context are combined together so if the user creates a new contact entry for this researcher, the researcher's name, e-mail address, phone number, address and home page entries will be automatically filled (Figure 6).

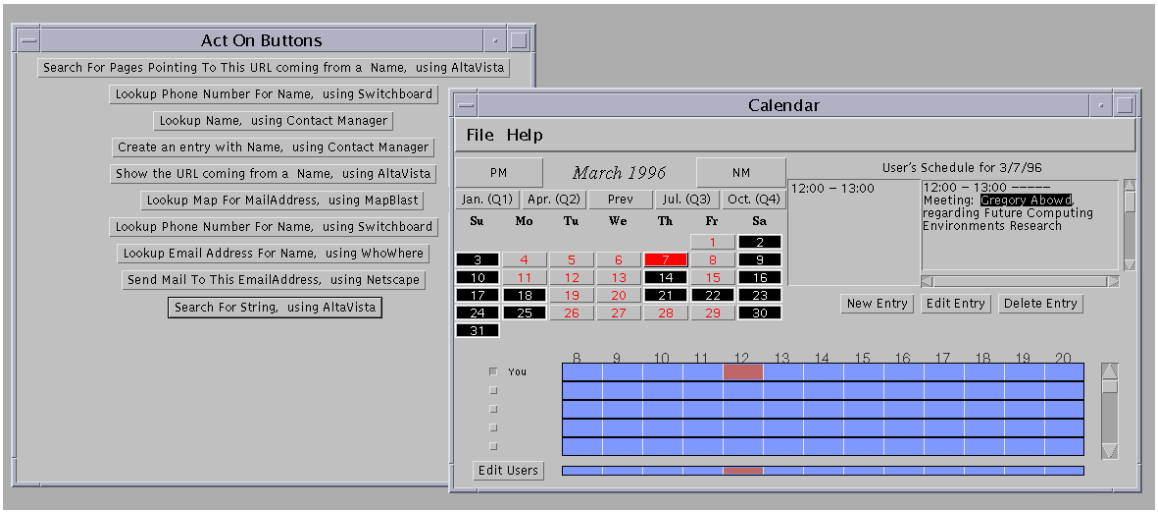


Figure 5: CyberDesk screenshot where services are offered based on context that was interpreted and aggregated.

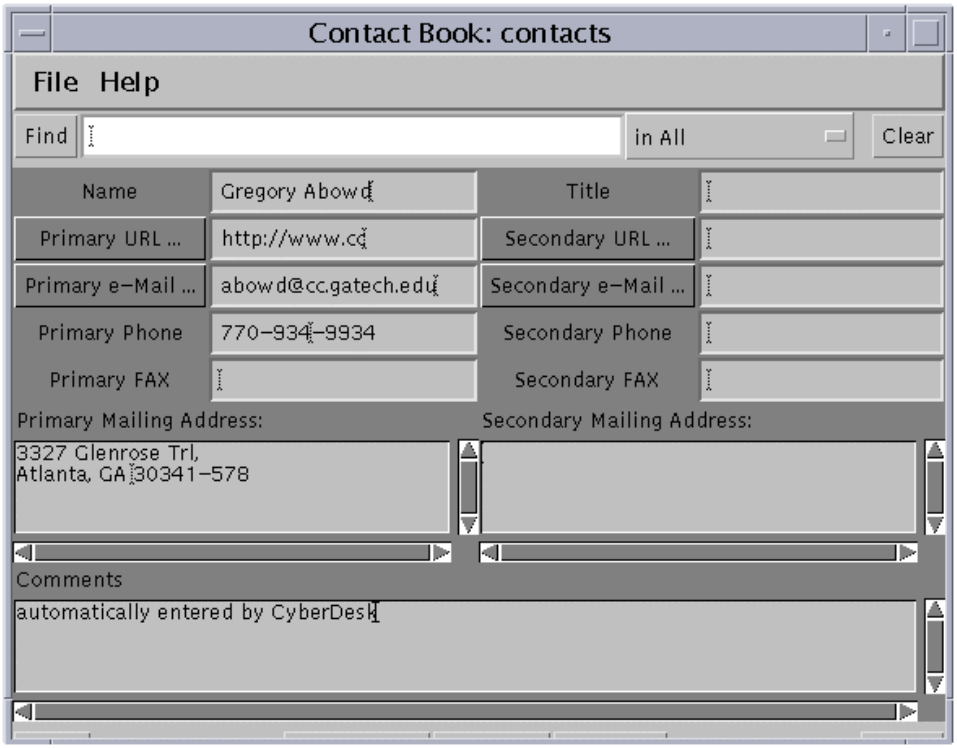


Figure 6: CyberDesk screenshot where a service has been performed on interpreted and aggregated context.

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. We determined these shortcomings when we attempted to use it to build an intelligent environment application (Dey 1998).

3.4.2.5 EasyLiving

EasyLiving is research project that is investigating appropriate architectures and technologies for intelligent environments (Brumitt, Shafer *et al.* 1999; Brumitt, Meyers *et al.* 2000). The goal of this project is to develop an architecture that will support a coherent user experience as users interact with a variety of devices in the intelligent environment. Software components are used to represent and encapsulate input and output devices. They register with a lookup service to allow other components and applications to locate them when necessary. A middleware called InConcert has been developed to support communications between distributed components, using asynchronous messages constructed in XML (eXensible Markup Language). A major focus of the EasyLiving project is to accurately model the geometry of the intelligent environment to better inform applications of physical relationships between people and devices and locations. Example applications developed as a part of EasyLiving include a dynamic remote control that provides access to services that are co-located with the user, teleporting of displays to the user's current location, personalizing media devices to play audio or video in the user's location that matches the user's interests. The architecture developed in EasyLiving uses InConcert and individual software components to provide support for separation of concerns, transparent communications and constant availability of components to multiple applications. There is partial support for specifying what context a component or application is interested in and for resource discovery. There is no explicit support for interpretation or context storage.

3.4.2.6 Schilit's System Architecture

In his Ph.D. thesis, Schilit presented a system architecture that supported context-aware mobile computing (Schilit 1995). This work has been very influential to our own research, helping 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. From our survey of context-aware computing (Dey and Abowd 2000b), we have seen that designers are indeed now capable of building context-aware applications, thanks in a large part to Schilit's work. He and his research group at Xerox PARC built a number of context-aware applications that focused mainly on users' locations, including locating the nearest printer, playing an audio file of a rooster whenever anyone in the building uses the coffee machine and displaying an important message on a display close to the user.

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: device agents that maintain status and capabilities of devices; user agents that maintain user preferences; and, active maps that maintain the location information of devices and users. 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, similar to the problems seen in 2.2.2. This makes it very difficult to evolve existing applications.

This work has a limited notion of context and does not include time or activity information. To add the use of other types of context, the user and device agents would have to be rewritten, making it difficult to add new sensors and context. The architecture supports the delivery of context through efficient querying and notification mechanisms. For reception, the architecture supports a limited notion of discovery, allowing applications to find components that they are interested in. However, applications must explicitly locate

these components before they can query or subscribe to them for context information. This is an accidental step that our simpler design process removes. Interpretation of context is not supported requiring applications to provide their own support. Finally, the lack of time information combined with the lack of context storage, makes it impossible for applications to acquire previous context information. This limits the amount of analysis an application can perform on context, which is an integral part of performing context-aware actions.

3.4.2.7 CALAIS

CALAIS, the Context And Location Aware Information Service, 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. It focused almost exclusively on location information, using video cameras, Active Badges, motion detectors, door sensors (to sense openings and closings) and keyboard monitors. It used the location information to identify users and devices in particular rooms, to track users as they move throughout an instrumented environment, to locate the nearest device with particular attributes (*e.g.* a printer) and to support teleporting.

An abstraction with a uniform interface was developed to hide the details of sensors from context-aware applications, with CORBA being used to provide the layer of separation (Object Management Group 2000). However there was very little support to aid developers in adding new sensors to the architecture. Additionally, 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 occurred with no intervening events. This is a powerful mechanism that makes the acquisition and analysis of context easier for application developers.

3.4.2.8 Technology for Enabling Awareness

The Technology for Enabling Awareness (TEA) project looked at how to provide awareness of context to personal mobile devices (Schmidt, Aidoo *et al.* 1999). It uses a blackboard model that allows *cues* (representations of sensors) to write their acquired context to it, *fact abstractors* (interpreters of context) to read context and write interpreted or abstracted context, and applications to read relevant context from it. This architecture was used to determine the state of a cell phone in order to automatically set its profile. For example, if the user is walking in an outdoor environment, the cell phone is put in a particular mode so that the volume is set to maximum and the vibrator is turned on. If the cell phone (and its owner) is thought to be in a meeting, the cell phone is put into a silent mode, where all calls are directed to voicemail. The cues and fact abstractors provide a separation of concerns between how context is acquired and how it is used by applications. There is limited support for specifying what context an abstractor or application is interested in. However, there is almost no support for transparent communications, constant availability to multiple applications or storage.

3.4.3 Proposed Systems

In this section, we present four more architectures for supporting the building and execution of context-aware applications. These architectures have been proposed with very little or no implementation.

3.4.3.1 Situated Computing Service

The proposed Situated Computing Service has an architecture that is similar to CyberDesk for supporting context-aware applications (Hull, Neaves *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. This makes it difficult to add new sensors to the system. It provides little support for allowing an application to specify what context it is interested in, to interpret context or to discover new sources of context. Finally, there is no support for the storage of context in the server, limiting the server's ability to perform complex analysis on the acquired context data.

3.4.3.2 Human-Centered Interaction Architecture

The Human-Centered Interaction Architecture is a proposed architecture that supports multimodal input involving multiple people in an aware environment (Winograd 2001). It is an attempt to view computing systems as collections of user-centered models, instead of as collections of devices. The architecture contains elements of abstraction, so applications do not have to deal directly with the details of sensors and actuators. Additionally, observer components can acquire data from sensors, and have a standard set of properties and events to provide a uniform interface to other components. Some observer components have the ability to translate between data types, although these data types appear to be focused on explicit input provided by users. Additionally, there are application-specific and person-specific models that can be used by both applications and the architecture to perform services and infer higher-level knowledge. While the goal of this work is to aid in collaborative multimodal computing, the qualities of abstraction and higher-level inferencing apply to the field of context-aware computing. It is a distributed blackboard-based architecture, allowing for simple separation of concerns, constant availability of context handling components and transparent distributed communications. However, there is no explicit support for storage, nor does it support a sophisticated mechanism for specifying what context an application is interested in.

3.4.3.3 Context Information Service

The Context Information Service (CIS) 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 relationships such as "close to". For example, the set of nearby printers would be specified by a "close to" relationship 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. There is no indication how any of these proposed features will be implemented. There is no explicit support for adding new sensors and context types, distributed communications, context storage, resource discovery or constant availability of context handling components.

3.4.3.4 Ektara

Ektara is a conceptual wearable computing architecture that will aid in managing a user's attention (DeVaul and Pentland 2000). The main piece of this proposed architecture is a *context-aware interaction manager* whose goal is to minimize the demands on the user's time and attention while maximizing the relevance of the information delivered to the user. Context will be acquired by sensors and classified or interpreted using a *perceptual context engine* that will use both a signal processing system and an inferencing system. Context can be stored and retrieved using a distributed database called the *contextual information service*. An initial version of Ektara has been implemented, supporting such applications as automatically logging a user into a nearby computer and creating and delivering location-based reminders. Ektara is intended to support more complex applications such as waiting until the user is not busy to deliver information and messages, allow the creation and delivery of arbitrarily complex reminders and assisting in shopping (both in directing the user to a store with good prices and in authorizing transactions). As stated earlier, Ektara is a conceptual architecture that is only now being implemented. It does not provide much support for separation of concerns or adding new sensors or interpretation of sensor data. Support for context storage is

limited to currently valid context, where outdated context can expire and be removed. This makes it difficult to perform an analysis of historical context. There is some support for specifying what context an application or the perceptual context engine is interested in, for transparent communications and constant availability, but because the architecture is conceptual, it is difficult to evaluate how much support there is for each of these features.

3.4.4 Existing Architectures Summary

Table 2 lists the architectural features that are supported in existing frameworks and will be supported in proposed frameworks. None of these frameworks support all the necessary features for supporting context-aware applications. In the next section, we will present our conceptual framework that supports all of these required features and the design process discussed earlier in this chapter.

Table 2: Feature Support in Existing and Proposed Architectural Support.

CS = context specification, SEP = separation of concerns, I = interpretation, TDC = transparent distributed communications, CA = constant availability, ST = storage and RD = resource discovery.
✗ = no support, P = partial support, ✔ = complete support

System	CS	SEP	I	TDC	CA	ST	RD
Open Agent Architecture	P	✔	✔	✔	✗	✗	P
Hive	P	P	✗	✔	✔	✗	✔
MetaGlue	P	✔	✗	✔	✔	✗	✔
Stick-e Notes	P	✔	✗	✗	✗	✗	P
Sulawesi	P	P	✗	✗	✔	✗	P
CyberDesk	P	✔	✔	P	✗	✗	P
EasyLiving	P	✔	P	✔	✔	✗	P
Schilit's System	P	P	✗	P	✔	✗	P
CALAIS	✔	P	✗	✔	✔	✗	✔
TEA	P	✔	✔	✗	✗	✗	P
Situated Computing	P	P	P	P	P	✗	P
Human-Centered Interaction	P	✔	P	✔	✔	✗	P
Context Information Service	P	✗	P	✗	✗	✗	✗
Ektara	P	✗	P	P	P	P	✔

3.5 *Architectural Building Blocks*

In the previous section (3.3), we discussed the basic features that are required to support library programmers. In this section, we will present features that are required to support toolkit programmers, those users of the architecture that will modify the existing architecture or populate the library of components. In most of our previous discussions, we have assumed that all the necessary building blocks and services were available. If we needed a component that provides identity information or an interpreter that converts iButton docking events to in/out status, we assumed that such a component already existed and we used it. But what happens if such a component does not exist? Unlike typical graphical user interface toolkits, where all of the basic widgets are already available for use (*e.g.* the original 7 Macintosh interactors), a "context toolkit" cannot provide all of the basic components. The variety of context that can be sensed, the variety of ways each piece of context can be sensed, and the variety of ways of interpreting context makes it impossible (at least at this early stage of investigating context-aware computing) to provide all context sensing and interpreting components *a priori* to an architecture user. Therefore, the architecture must support developers in building their own context components that can be used within the framework of the existing architecture.

It is left up to the application designer whether to implement the functionality either within the application or as a separate component. Clearly, we prefer the second of these alternatives. Implementing the functionality as a separate component allows reuse by other applications and other application designers. To encourage the choice of the second alternative, we need to provide support so that the building of these components is actually easier than *ad hoc* implementation within the application. By minimizing the effort required to construct the building blocks, we can provide this support. What this means is that when a developer wants to build one of these components, the relevant features should already be implemented and automatically available for her leverage from. This leaves the developer with the much simpler task of designing only the task-specific or *essential* pieces of the component. If we provide a standard interface for each type of building block, it will allow applications and the rest of the framework to deal and communicate with each of these building blocks in a similar fashion.

Previously, we have discussed two types of building blocks, one for acquiring context from sensors that we will refer to as a *context widget*, and another that interprets context that we will refer to as a *context interpreter*. We will provide more details on these components as well as introduce two new building blocks, one that collects related context together that we will refer to as a *context aggregator*, and one that provides reusable context-aware behaviors or services to applications that we will refer to as a *context service*.

3.5.1 Context Widgets

Context widgets are responsible for collecting information from the environment through the use of software or hardware-based sensors (Salber, Dey *et al.* 1999a). They are named after GUI widgets and share many similarities with them. Just as GUI widgets mediate between the application and the user, context widgets mediate between the application and its operating environment. Both types of widgets provide an abstraction that allow events to be placed in and taken out of the event queue without applications and widgets having to know the details about each other. Where GUI widgets are owned by the application that creates them, context widgets are not owned by any single application and are shared among all executing applications. We will analyze the benefits of GUI widgets, introduce the concept of a context widget, detail its benefits, and explain how developers can build context widgets.

3.5.1.1 Learning From Graphical User Interface Widgets

It is now taken for granted that GUI application designers and programmers can reuse existing interaction solutions embodied in GUI toolkits and widget libraries. GUI widgets (sometimes called interactors) span a large range of interaction solutions: selecting a file; triggering an action; choosing options; or even direct manipulation of graphical objects (Myers 1990).

GUI toolkits have three main benefits:

- They hide specifics of physical interaction devices from the applications programmer so that those devices can change with minimal impact on applications. Whether the user points and clicks with a mouse or fingers and taps on a touchpad or uses keyboard shortcuts does not require any changes to the application.
- They manage the details of the interaction to provide applications with relevant results of user actions. Widget-specific dialogue is handled by the widget itself, and the application often only needs to implement a single callback to be notified of the result of an interaction sequence.
- They provide reusable building blocks of presentation to be defined once and reused, combined, and/or tailored for use in many applications. Widgets provide encapsulation of appearance and behavior. The programmer does not need to know the inner workings of a widget to use it.

Although toolkits and widgets are known to have limitations such as being too low-level or lacking flexibility (Johnson 1992), they provide stepping stones for designing and building user interfaces and developing tools such as User Interface Management Systems (UIMS) (Olsen 1992). With context widgets, we aim at providing similar stepping stones for designing and building context-aware applications.

3.5.1.2 Benefits of Context Widgets

A context widget is a software component that provides applications with access to context information from their operating environment. In the same way GUI widgets insulate applications from some presentation concerns, context widgets insulate applications from context acquisition concerns, by wrapping sensors with a uniform interface.

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 presence of people is sensed using Active Badges, floor sensors, video image processing or a combination of these should not impact the design of 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 does not report less significant moves to the application. Widgets provide abstracted information that we expect applications to need most frequently.
- They provide easy access to context data through querying and notification mechanisms available through a common, uniform interface for accessing context. No matter what kind of context is being sensed from the environment, all widgets make it available in the same manner.
- 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 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.

3.5.1.3 Building Context Widgets

Context widgets have a state and a set of behaviors or callbacks. The widget state is a set of attributes that can be queried by applications. For example, a Presence widget has attributes for its location, the time a presence is detected, and the identity of the detected user. Applications can also register to be notified of context changes detected by the widget. The widget triggers callbacks to the application when changes in the environment are detected. The Presence widget for instance, provides callbacks to notify the application when a new person arrives, or when a person leaves.

To build a widget, a developer must specify what the widget's attributes and callbacks are. Often, a widget will have an attribute that has a constant value. For example, our Presence widget has a constant value for the location attribute. The location that this widget instance is responsible for is not changing. It is this combination of attributes with varying values, constant attribute values, and callbacks that defines a widget instance. Additionally, a developer must provide the connection to the underlying sensor technology. This means providing code that will allow both querying of the sensor and notification when the sensor's status changes from the widget level. To create a new context widget, a developer need only specify those portions of the widget that are specific to the particular widget instance being created. Widgets automatically support the following framework features: distributed communications and a global timeclock, common interface for querying and notification, separating the acquisition of context from how it is used, independence from individual applications, context storage, and allowing applications and other components to discover them.

3.5.2 Context Interpreters

Context interpreters are responsible for implementing the interpretation abstraction discussed in the requirements section. Interpreters abstract raw or low-level context information into richer or higher level forms of information. A simple example of an interpreter is one that converts a room location into a building location (e.g. Room 343 maps to Building A). An application may require the location in the form

of a building name rather than the more detailed room form. A more complex example is one that takes location, identity and sound information from a number of widgets in a conference room and infers that a meeting is occurring. Context interpreters can be as simple or as complex as the designer wants, from using a simple lookup table to using complex artificial intelligence-based inferences.

Interpretation of context has usually been performed by applications. By separating the interpretation abstraction from applications, we allow reuse of interpreters by multiple applications. All interpreters share a common interface so other components can easily determine what interpretation capabilities a given interpreter provides and know how to communicate with all interpreters.

To build a context interpreter, a developer must provide the parts of the interpreter that are specific to the instance being built. These include the context attributes that are to be interpreted, the context attributes that are the result of the interpretation, and the actual code that will perform the interpretation. Interpreters automatically provide the following architecture features to interpreter developers: transparent distributed communications, independence from applications, allowing applications and other components to discover them and a common interface that allows all applications to deal with all interpreters in the same way.

3.5.3 Context Aggregation

To facilitate the building of context-aware applications, our framework should support the aggregation of context about entities in the environment. 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. It is often the case that an application requires multiple pieces of information about a single entity. With the conceptual framework described so far, an application (via the context specification mechanism) would have to communicate with several different context widgets to collect the necessary context about an interesting entity. This adds complexity to the design and negatively impacts maintainability. Building blocks called context aggregators aid the framework in supporting the delivery of specified context to an application, by collecting related context about an entity that the application is interested in.

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. For example, imagine an extension to the In/Out Board that supported the use of multiple iButton readers installed in multiple buildings. The In/Out Board for a particular building is not interested where a dock occurred in the building, only in which building it occurred. Rather than communicate with each individual reader, the application could request notifications from the aggregator for each building, simplifying the application development.

To build a context aggregator, a developer simply has to specify the entity type and identity. Entity type refers to whether the aggregator represents a person entity, location entity or thing (device or object) entity. For example an aggregator that represents “Mary Smith” would have the type set to “user” and identity set to “Mary Smith”. All of the framework features common to aggregators are automatically provided for aggregator developers: separation of how context is acquired from how it is used (at a level higher than with widgets), transparent distributed communications, independent, persistent execution from applications, context storage and allowing applications and other components to easily discover them.

3.5.4 Context-Aware Services

From our review of context-aware applications, we have identified three categories of context-aware behaviors or services (section 1.2.3). 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 framework 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). 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.

To build a context service, a developer does not need to worry about providing a mechanism for calling and executing the service, dealing with transparent distributed communications, separation from applications, and allowing applications to discover it, but instead receives this functionality automatically. Instead, the developer must provide a description of the service's functionality, the set of parameters, if any, that the service requires to be executed (*e.g.* the message and potential responses to page a user with, the user's pager number), and code that interacts with an actuator to perform the service.

3.6 Building Context-Aware Applications with Architectural Support

Let us revisit the In/Out Board application with the new design process and the four building blocks just described. We will show how to apply the new design process and will show that we can build the application more easily by using the conceptual framework. First we will discuss how the application would be built if all the necessary building block instances were already available. Then we will discuss how the building blocks would be built if they did not already exist.

3.6.1 In/Out Board with Context Components

In the first step of the simplified design process, specification, the developer specifies the context-aware behavior to implement and the context required. In this case, the behavior is to display the in/out status of occupants of a building and when they were last seen (step 1.1). The context required is location, identity, in/out status and time. By using the resource discovery mechanism at design time, we see that all the necessary components are available (step 1.2). A widget that can deliver location, time and an iButton id and an interpreter that can convert from the id to a user identity are available, so no components have to be created or installed (step 2). The application requests to be notified (by subscribing to a widget callback) whenever a building occupant arrives or leaves the building, indicating that it wants the relevant time, user identity and in/out status in each notification. When the application receives such a notification, it performs its context-aware behavior, updating its internal status for the user whose status has changed and updating its display to match the new status information (step 3). When compared with the application design discussed in section 3.2.1, this process is much easier to follow and perform.

3.6.2 Building the Context Components Needed by the In/Out Board

In our discussion of the In/Out Board design, we assumed that all necessary context components were already available and executing in the environment. If neither the needed interpreter nor widget were available, the application designer would be responsible for providing these components. To build the interpreter, the designer specifies the incoming attribute, iButton id, and the outgoing attribute, user identity, and provides code that can perform the conversion. The code could be a hashtable with iButton ids

and user names stored in memory or something that will read a file that contains the ids and corresponding names.

To create a context widget that represents the iButton reader, the following tasks must be completed: specify the attributes the widget is responsible for, specify the callbacks the widget provides and provide support for querying the sensor state and being notified of changes in the sensor state. The attributes for the widget are iButton id, timestamp, in/out status and location (set to a constant location equal to the name of the building at instantiation time). The callbacks the widget provides to subscribers are in events, out events and both events. The iButton API provides support for querying and notification, so the developer just has to connect the existing API to the widget interface to allow applications to deal with a standard widget as opposed to a non-standard sensor. Both the creation of the interpreter and the widget have been simplified greatly through the provision of standard context components that provide all common functionality required.

3.7 Summary of the Requirements

This chapter described the requirements for a conceptual framework that supports the building, execution and evolution of context-aware applications, in order to foster the building of applications that use a variety of sensors and a variety of context and are easy to evolve. These requirements are:

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

In addition, this chapter introduced a complex design process for building context-aware applications. By analyzing the design process, it was shown that this design process could be simplified greatly by using an architecture that met the above requirements, removing all accidental steps from the process. The simplified design process follows:

1. *Specification*: Specify the problem being addressed and a high-level solution.
   1.1. Specify the context-aware behaviors to implement.
   1.2. Determine what context is required for these behaviors (with a knowledge of what is available from the environment) and request it.
2. *Acquisition*: Determine what hardware or sensors are available to provide that context and install them.
3. *Action*: Choose and perform context-aware behavior.

Finally, architectural building blocks were introduced that help support the design process and fulfill the architectural requirements. These building blocks are context widgets that abstract the details of sensing or acquiring context from using it, context interpreters that transform between different types of context, context aggregators that collect related context for particular entities (people, places, or things) and context services that perform context-aware behaviors on behalf of applications. These components support a programming model known as the *context component* programming model.

CHAPTER 4

IMPLEMENTATION OF THE CONTEXT TOOLKIT

In the previous chapters we have introduced the requirements for a conceptual framework that supports the building and execution of context-aware applications and have also defined a design process for building these applications. In this chapter we will discuss an implementation of the framework we have laid out, called the Context Toolkit (Dey, Salber *et al.* 1999; Salber, Dey *et al.* 1999a; Dey, Salber *et al.* 2001). We will describe the Context Toolkit, its components and how it supports the requirements and design process presented earlier. See APPENDIX A for more information on the Context Toolkit, including documentation and download information.

The Context Toolkit consists of two parts: an embodiment of the previously defined architecture for supporting the building and execution of context-aware applications and a provided library of widgets, aggregators, interpreters and services. The toolkit was primarily written in Java (with a handful of widgets and applications written in other languages). See APPENDIX B for a list of all the components and applications and the languages they were written in. The toolkit consists of roughly 29000 lines of code (at the time of this publication). In this discussion of the toolkit, we will describe how applications use the architecture, the components in the toolkit and how they support the design process and required features for a supporting architecture.

4.1 Components in the Context Toolkit

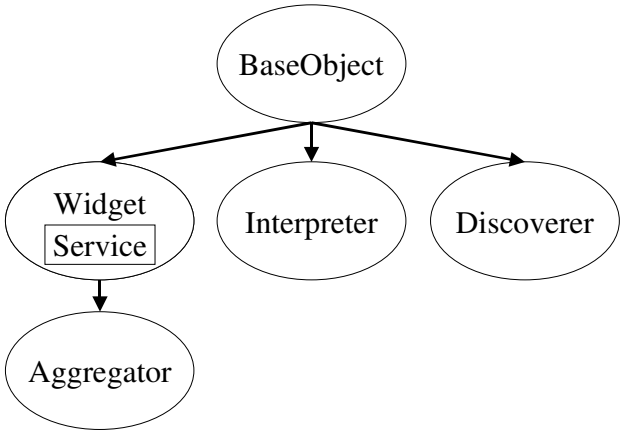


Figure 7: Context Toolkit component object hierarchy where arrows indicate a subclass relationship

Not surprisingly, the components used in the Context Toolkit almost mirror the components described in the conceptual framework (3.5). In the Context Toolkit, we have a basic communications component, known as the BaseObject, context widgets, context aggregators, context interpreters, context services (hereafter referred to as widgets, aggregators, interpreters and services, respectively) and resource discovery objects, known as discoverers. These components make up the *context component abstraction*, a conceptual framework that makes it easier to design, build and evolve context-aware applications. The

framework allows developers to think of building an application in terms of logical building blocks, objects that acquire, collect, transform, deliver and act on context. An object hierarchy diagram of the components that make up this framework is given in Figure 7. Widgets, interpreters and discoverers subclass from BaseObject and inherit its communications functionality. Services are part of the Widget object, in the same way that callbacks and attributes are (see Section 4.1.3). Finally, aggregators subclass from widgets, inheriting all the functionality of widgets.

Figure 8 shows how applications and components in the Context Toolkit can interact with each other. When widgets, aggregators and interpreters are instantiated, they register themselves with a Discoverer. When an application is started, it contacts the Discoverer to locate components that are relevant to its functionality. For example, the In/Out Board application uses a Discoverer to find Location Widgets that can tell it about the location of occupants in the building that it installed in. Widgets collect context from sensors and make it available to both aggregators and applications. They also provide services that applications can execute. Aggregators collect context about particular entities by subscribing to widgets that can provide this context. Applications can also be subscribers to widgets or can acquire context from widgets by querying them. Applications do not have to acquire context only from widgets, but can also subscribe to or query aggregators. Finally, widgets, aggregators and applications can request that context be transformed using interpreters.

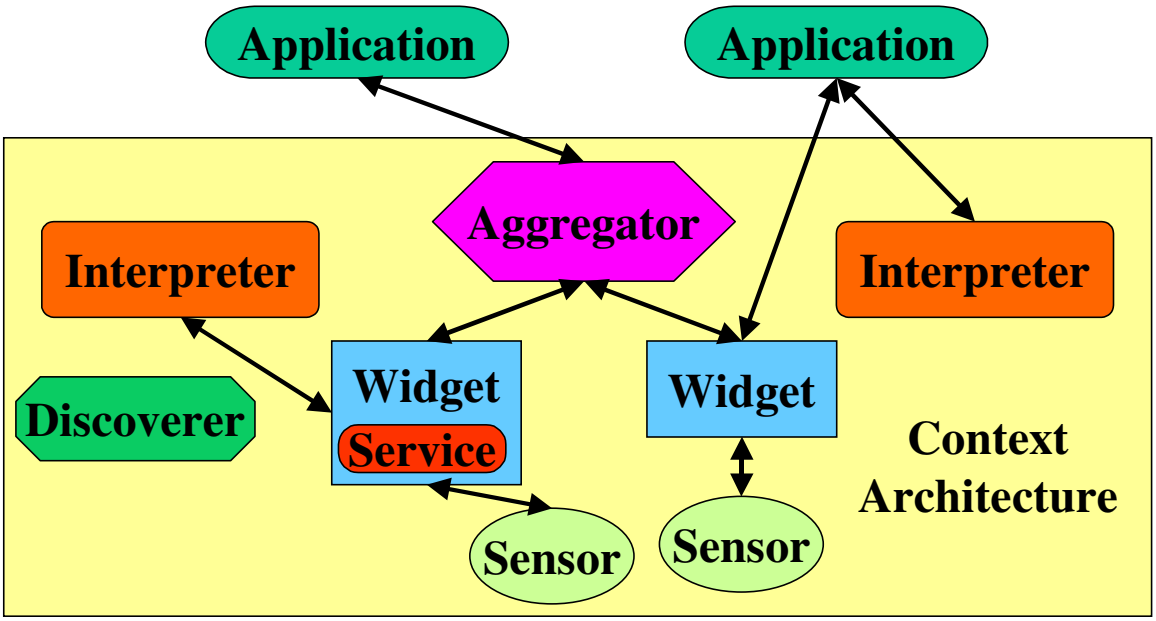


Figure 8: Typical interaction between applications and components in the Context Toolkit

In the remainder of this sub-section, we will discuss these components in detail and describe how they are used by programmers.

4.1.1 BaseObject

The BaseObject class provides the basic communications infrastructure needed to communicate with the distributed components in the Context Toolkit. It facilitates the peer-to-peer communications used in the toolkit, being able to send, receive, initiate and respond to communications. Programmers do not write BaseObjects but instead create instances of them to use. Applications use BaseObject instances to communicate with the context-sensing infrastructure and components within the infrastructure sub-class from BaseObject to allow them to communicate with other infrastructure components.

BaseObject contains a number of methods for communicating with widgets, aggregators, interpreters, services and discoverers. We will discuss these methods in the sections for each of these components. For these outgoing communications, where BaseObject behaves as a client, data provided in the Context Toolkit-specific format (defined below) is first encoded using the eXtensible Markup Language (XML) version 1.0 (Bray 2000) and then wrapped with the HyperText Transfer Protocol (HTTP) version 1.0 (Berners-Lee, Fielding *et al.* 1996) to allow transfer to another Context Toolkit component or context-aware application. To communicate with any context component, three parameters must be provided: hostname (or I.P. address) of the machine the component is executing on, the port number the component is listening for communications on and the unique identity of the component. These parameters do not have to be known *a priori* by the object using a BaseObject instance, but can be determined at runtime using a Discoverer (discussed in Section 4.1.4). The first two parameters specify the network location of the component. The identity is used to ensure that the message is being delivered to the correct component. There are some methods that BaseObject supports for communicating with all types of components in the infrastructure (Figure 9). This includes a method that "pings" another component to ensure that it is still operating and one that queries for the version of a remote component.

```
Client:     pingComponent: ping a component
            queryVersion: request a component's version

Server:     pingComponentReply: response to a ping
            queryVersionReply:  response  containing  version
                                number
```

Figure 9: BaseObject's client and server message types.

When any component in the Context Toolkit receives a communications message, it, as a subclass of BaseObject, behaves like a server. For example, when a component receives a ping request, it automatically responds to the requestor indicating that it is indeed available for further communications. When a component receives a query for its version, it also automatically responds to the requestor with its version number. To handle simultaneous incoming communications, BaseObject uses a multithreaded server. When it receives information, the server's current thread handles the new request and it forks a new thread to listen for future communications. The current thread first strips the network protocol from the request and then decodes the request into a Context Toolkit-specific format. The information in the request is then examined so the request can be handled appropriately. HTTP and XML are provided as default implementations for the network protocol and message language.

Additionally, BaseObject is responsible for maintaining a global timeclock. Each instance automatically connects to a Simple Network Time Protocol (SNTP) Version 4 (Mills 1996) server in order to synchronize its local time with a standard global time. Synchronization occurs at regular intervals to ensure that all components and applications have a common time with which to compare data.

Outgoing messages are encoded in XML and HTTP and incoming messages are stripped of the HTTP and XML encodings. The format underlying the XML and HTTP is a Context Toolkit-specific data structure called DataObject. Figure 10 describes the representation of DataObject.

```
DataObject = {name, Data}
Data = value | DataObject*
```

Figure 10: Context Toolkit internal data format for messages.

Figure 11 gives a concrete example of a `queryVersion` message and response with the additional XML and HTTP encodings. A complete list of all the valid messages in the Context Toolkit is given in APPENDIX C. In this example, a component running on the machine "quixotic.cc.gatech.edu" wants to

know the version number of a component named “widgetId”. The queried component, running on “munch.cc.gatech.edu”, responds with a version number of “1.0” with no errors.

```
queryVersion DataObject =
       {“queryVersion”, {“id”, “widgetId”}}

queryVersion DataObject with XML encoding =
       <?xml version=”1.0”?>
       <queryVersion>
             <id> widgetId </id>
       </queryVersion>

queryVersion DataObject with XML and HTTP encoding =
       POST queryVersion HTTP/1.0
       User-Agent: Context Client
       Host: quixotic.cc.gatech.edu
       Content-Type: text/xml
       Content-Length: 80

       <?xml version=”1.0”?>
       <queryVersion>
             <id> widgetId </id>
       </queryVersion>


queryVersionReply DataObject =
       {“queryVersionReply”,
       {{“version”, “1.0”}, {“errorCode”, “noError”}}

queryVersionReply DataObject with XML encoding =
       <?xml version=”1.0”?>
       <queryVersionReply>
             <version> 1.0 </version>
             <errorCode> noError </errorCode>
       </queryVersionReply>

queryVersion DataObject with XML and HTTP encoding =
       HTTP/1.0 200 OK
       Date: Sun Oct 28 17:25:31 EST 2000
       Server: context/1.0
       Content-type: text/xml
       Content-length: 126

       <?xml version=”1.0”?>
       <queryVersionReply>
             <version> 1.0 </version>
             <errorCode> noError </errorCode>
       </queryVersionReply>
```

Figure 11: Example queryVersion message and response showing internal and external formatting.

Internally, a message is represented as a DataObject, which consists of a name and some data. The name of a DataObject for an outgoing message is just the name of the request being made (*e.g.* `queryVersion` in

Figure 11) and the name of the DataObject for an incoming message is the name of the request with the word "Reply" appended to it (*e.g.* `queryVersionReply` in Figure 11). The data can be either a single value (*e.g.* "widgetId" for the "id" DataObject) or a vector of DataObjects (*e.g.* {{"version", "1.0"}, {"errorCode", "noError"}}). When a request is made to another component, the network socket connection between the two components is maintained until the component replies. Once the reply is received, the connection is broken.

4.1.2 Widgets

Widgets are used to separate how context is acquired from how it is used. They acquire context from sensors and provide a uniform interface to applications that want to use the context. Applications can subscribe to widget callbacks to be notified of changes to the widgets' context information or can query widgets to get the latest context information. In addition, widgets store the context they acquire so applications can leverage off historical information. They typically run on the machines that their sensors are connected to, whether they be desktop machines, embedded systems, or mobile devices.

A basic Widget object that sub-classes from BaseObject is provided in the Context Toolkit, handling all the features that are common across widgets. It is responsible for maintaining a list of subscribers to each callback in the widget, storing all data that is sensed, mediating access to the widget's services and allowing inspection of the widget's internal information.

Because Widget sub-classes from BaseObject, it automatically inherits the communications functionalities, being able to act as a client or a server. Most commonly, widgets behave as servers that deliver requested information to clients. When an incoming message is received by a widget, BaseObject tries to handle it. If BaseObject is unable to handle it (*i.e.* if it is not a `pingComponent` or `queryVersion` message), the widget handles the message and responds. Widgets can handle the following types of messages or requests:

Widget inspection

- `queryAttributes`: Request for a list of the context attributes the widget can provide
- `queryCallbacks`: Request for a list of the callbacks the widget can provide notification about
- `queryServices`: Request for a list of the services a widget can perform

Widget subscriptions

- `addSubscriber`: Request to subscribe to a widget callback
- `removeSubscriber`: Request to unsubscribe to a widget callback

Widget storage

- `query`: Query the widget for its last acquired value
- `updateAndQuery`: Request that the widget acquire new data and return it
- `retrieveData`: Request to retrieve data from the widget's persistent storage

Widget services

- `serviceRequest`: Request to execute a widget's asynchronous or synchronous service

An application (or another component) can make these requests via an instance of BaseObject.

4.1.2.1 Widget Inspection

The first three messages (`queryAttributes`, `queryCallbacks` and `queryServices`) allow an application to inspect a widget's capabilities. Commonly used Context Toolkit data structures and their representations are shown in Figure 12. Context data is represented as a collection of typed attribute name-value pairs. Values can be simple primitives or can be more complex structures consisting of nested

attribute name-value pairs. Attributes represent the types of context that a widget can provide. AttributeNameValues are similar, but represent actual context data. Callbacks, or remote events, represent changes in the widget context that interested components can subscribe to and consist of a name and a set of attributes.

Attribute: name of an attribute and a data type (*e.g.* boolean, integer, short, long, float, double, string, struct)
Attribute example: `[identity, string]`
DataObject representation:
```
{{"attribute type=string",
  {"attributeName", "identity"}}
```

Attribute struct example:
```
[contactInformation, struct [[fullname, string]
                             [phone, string]]]
```
DataObject representation:
```
{"attribute type=struct",
 {{"attribute type=string", {"attributeName", "fullname"}},
  {"attribute type=string", {"attributeName", "phone"}}}
```

AttributeNameValue: Attribute with a value
Example: `[identity, string, "Anind"]`
DataObject representation:
```
{"attributenamevalue type=string",
  {{"attributeName", "identity"}
   {"attributeValue", "Anind"}}}
```

Callback: name of a callback and a vector of Attributes that will be returned with the callback
Example:
```
[locationUpdate, [[location, string]
                  [identity, string]
                  [timestamp, long]]]
```
DataObject representation:
```
{"callback",
 {{"callbackName", "locationUpdate"}
  {"attributes",
    {"attribute type=string", "location"},
    {"attribute type=string", "identity"},
    {"attribute type=string", "timestamp"}}}}
```

Condition: attribute name, operator (=, <, <=, >, >=), and a value
Example: `[identity, =, "Anind"]`
DataObject representation:
```
{"condition",
  {{"name", "identity"},
   {"compare", "="},
   {"value", "Anind"}}}
```

Figure 12: Context Toolkit data structures and their representations.

4.1.2.2 Widget Subscriptions

An application (or other component) can subscribe to a widget callback (*i.e.* add itself as a subscriber) using the `subscribeTo` method (Figure 13) provided in BaseObject. This method sends an `addSubscriber` message to the widget.

```
public Error subscribeTo(Handler handler, int port, String subid,
   String remoteHost, int remotePort, String remoteId,
   String callback, String tag, Attributes attributes,
   Conditions conditions)
```

Figure 13: Prototype for the subscribeTo method

A subscriber must specify the following information:

- hostname of the machine the subscriber is running on;
- handler object that will handle any callback notifications;
- port number the subscriber's BaseObject is listening for communications on;
- unique id of the subscriber;
- widget callback the subscriber is subscribed to;
- tag used by the subscriber to differentiate callbacks
- attributes the subscriber is interested in (optional); and,
- conditions the subscriber has placed upon the subscription (optional).

The specification of the callback, attributes and conditions essentially act together as a filter to control which data and under which conditions context events are sent from a widget to a subscribing component to be handled. Each widget callback has an associated set of context attributes, as illustrated in Figure 12. If a subscribing component is only interested in a subset of those attributes, it uses the attributes parameter to indicate which ones. For example, a widget callback may return three types of context data, but the subscriber may only be interested in two.

The conditions are used by a subscriber to indicate the circumstances in which the subscriber should be notified when the callback is fired. For example, a callback may be fired whenever the value of a context type changes, but the subscriber may only need to be notified when the value changes to a particular value or range. The subscriber can use the conditions parameter to specify this. Multiple conditions are logically AND'ed together. Currently OR and NOT are not supported.

This combination of the callback, attributes and conditions is an extension of the conventional subscription mechanism, where only callbacks can be specified. This helps to substantially reduce the amount of network communication, which is important in a distributed architecture for performance reasons. This mechanism also makes it easier for application programmers to deal with context events, by delivering only the specific information the application is interested in.

Each time a widget's sensor senses new data, the widget determines whether any subscribers need to be notified. It first detects whether the new data is significant – this is a widget-specific decision. If the data is significant then the relevant callback is chosen. For each subscription to this callback, the widget determines whether the data satisfies the specified conditions, if any. If there are no conditions or the conditions are satisfied, the data is filtered according to the specified attribute subset, if any, and then sent to the subscriber (using a `subscriptionCallback` message type). The subscriber's BaseObject receives the message and, if there are no errors in the message, replies with a `subscriptionCallbackReply` message.

When a component subscribes to a widget, it specifies a local object (that could be the component itself) known as a *handler*, that will be responsible for handling the callback data. If it subscribes to multiple

widget callbacks, it can specify the same handler for each subscription, or it can specify multiple ones. Each handler must implement the `Handler` interface, which has one method called `handle()` that will actually handle the callback message. A unique key is constructed from the subscription information that corresponds to a handler. Upon receiving the `subscriptionCallback message`, the subscriber's BaseObject instance uses the data from the callback to construct a key and determine which handler object to send the data to. It calls that object's `handle()` method.

A simple example demonstrating the subscription process for the In/Out Board application, introduced in CHAPTER 3, is shown in Figure 14. In this example, a Location widget is used to determine when an individual has entered or left the location it is responsible for, in this case, Building A. The In/Out Board locates the widget with location information about Building A using the Discoverer (a). It (b) subscribes to the Location Widget, wanting to know when Gregory has entered or exited Building A. Although Figure 14 and Figure 15 only show a single subscription, there would be a subscription for each individual the In/Out Board cared about (or the In/Out Board could just subscribe once for all location events and perform the filtering itself in determining which updates are updates about people it cares about). The widget adds the In/Out Board to its list of subscribers and replies with an error value for the subscription, indicating whether the subscription was successful or not, and if not, why. The network connection between the application's BaseObject and the widget is removed after the reply is received. When the Location Widget has new location information available (c), it compares the information to see if the information meets the subscription conditions. In this case, since the information is about Anind, the context is stored by the widget and is not sent to the application. When the Location Widget has new location information about Gregory (d), it sends the information to the application (e) and the application's BaseObject routes it to the `handle()` method (f). In this case, the application itself is acting as the handler for context information. Actual application code is given in Figure 15.

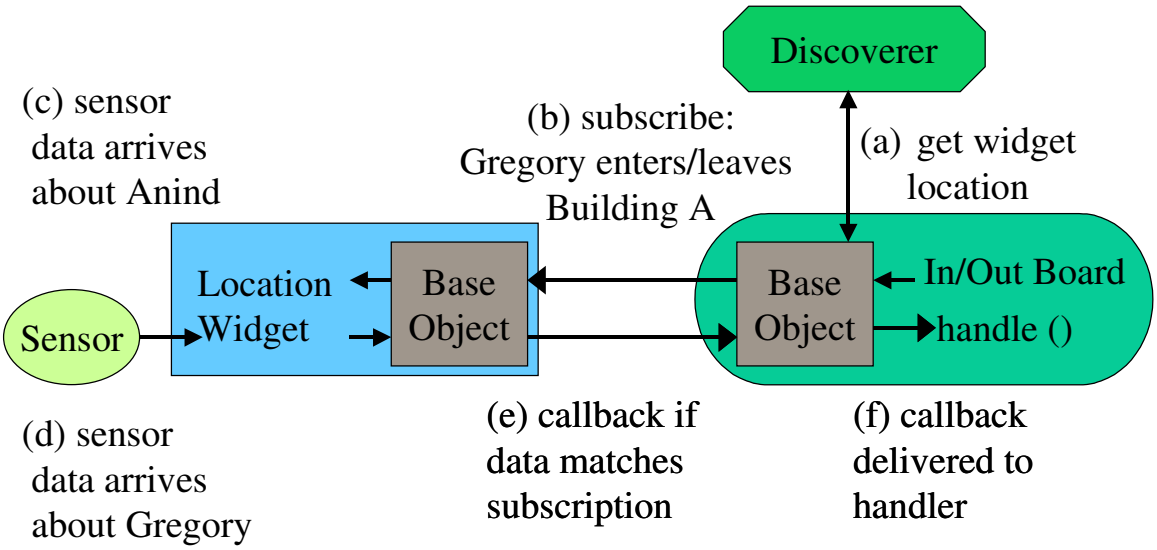


Figure 14: Example of an application subscribing to a context widget.

```
public class InOutBoard implements Handler {
        public InOutBoard(String location) {
                // create BaseObject to enable communications with infrastructure – receives
                // communications on port number defined by PORT
                BaseObject contextConnection = new BaseObject(PORT);

                // code to determine network location (widgetHost and widgetPort) and
                //  identity of Location Widget (widgetId) with information about Building A
                //  using the Discover (a)
                …

                // set the Attributes you want returned in the callback
                Attributes attributes = new Attributes();
                attributes.addAttribute(USERNAME);
                attributes.addAttribute(TIMESTAMP);
                attributes.addAttribute(IN_OUT);

                // code to subscribe (b) to widget for information about Gregory – repeat for each user
                // Sets this class as the handler for the callback information, subscribes to the widget's
                // locationUpdate callback, and sets a local callback of IOBupdate
                Conditions conditions = new Conditions();
                conditions.addCondition(USERNAME, Storage.EQUAL, "Gregory");
                Error error = contextConnection.subscribeTo(this, PORT, "InOutBoard", widgetHost,
                        widgetPort, widgetId, "locationUpdate", "IOBupdate", conditions, attributes);

                …
        }

        // called when data matches subscription (e, f)
        public DataObject handle(String callback, DataObject data) {
                if (callback.equals("IOBupdate")) {
                        AttributeNameValues data = new AttributeNameValues(data);
                        String user = (String)data.getAttributeNameValue(USERNAME).getValue();
                        String status = (String)data.getAttributeNameValue(IN_OUT).getValue();
                        Long time = (Long)data.getAttributeNameValue(TIMESTAMP).getValue();

                        // update the display with the received context
                        UpdateDisplay(user,status,time);
                }
        }

        …
}
```

Figure 15: Application code demonstrating a widget subscription and handling of the callback notification.

If the building has multiple sensors (*e.g.* face recognition system and an Active Badge system), there would be multiple widgets that could indicate whether a user was entering or leaving the building. The code given in Figure 15 would be identical except that there would be multiple `subscribeTo` method calls (one for each widget's callback). The diagram in Figure 14 would be similarly changed to that of Figure 16, with the application finding all the widgets that can provide location information about Building A and subscribing to each widget, and each widget being responsible for notifying the application when a relevant change occurred.

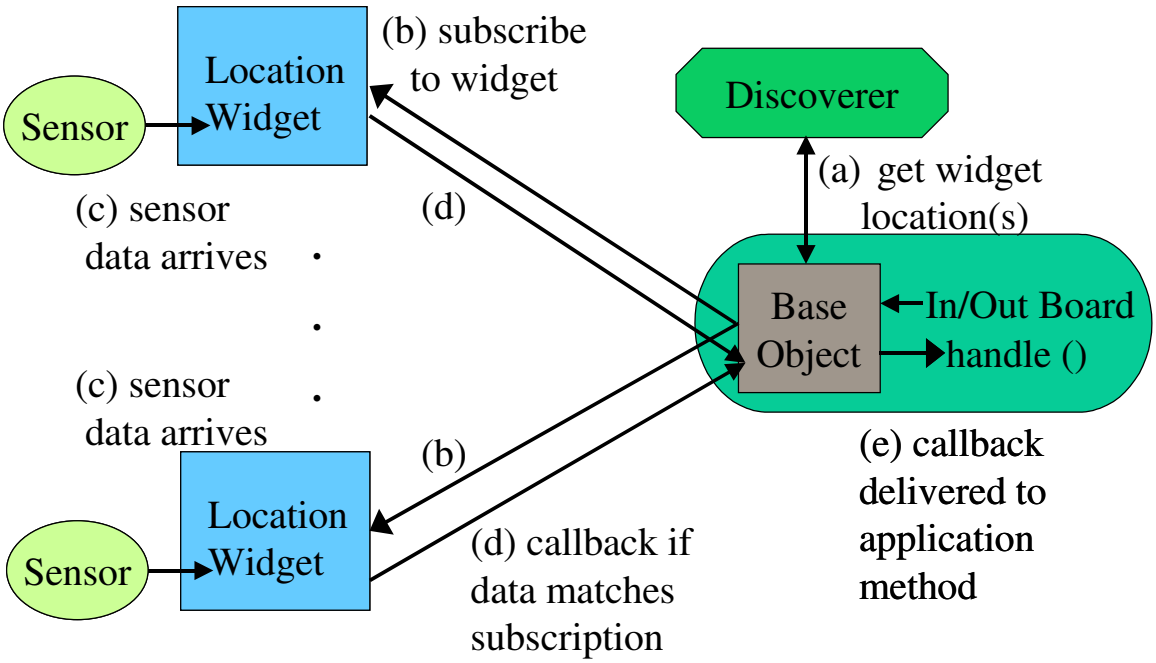


Figure 16: Example of an application subscribing to multiple context widgets.

As part of maintaining a list of its subscribers, a widget can optionally create a subscription log. This log keeps track of all the current subscribers to a widget. If a widget is stopped or the machine it is being executed on crashes, then when the widget is restarted it can use the log to reconnect to its subscribers. A subscriber is removed from the log and the list of subscribers when it uses the `unsubscribeFrom` method (sends a `removeSubscriber` message to the widget).

4.1.2.3 Widget Storage

By default, a widget automatically stores its data, with no code required from the programmer, and allows other components to access this context history. (However, at instantiation time, a variable can be set to turn storage off). Widgets have two mechanisms for storing data. The simplest is a local cache in memory that stores only recently sensed context. The local cache is used to quickly respond to requests for the latest data (`query`). An `updateAndQuery` request causes the widget to acquire new data from its sensor, put the data in its local cache and return it. Context data is flushed from the local cache to the second, and more persistent, store according to a specified flush policy. Whenever the widget gets new data from its sensor, the data is stored in the local cache. The widget then checks to see whether the flush condition has been reached. Supported policies are to flush whenever a specified number of pieces of context have been acquired from a sensor or to flush whenever a specified amount of time has passed. An application can acquire this historical context from the persistent store using a `retrieveData` request.

The default persistent store is a table in a remote mySQL (mySQL 2000) database (a freeware relational database), accessed using the Java Database Connectivity (JDBC) (Sun Microsystems 2000c) support. When a widget is created and instantiated, it automatically creates a table in the database (if it does not already exist). The table's name is the widget's id and the table columns are the widget's attributes. If all components use the same database, the configuration details can be set once and applied to all components. However, if components want to use a variety of databases or multiple copies of the same database (for reasons of distribution), a programmer can specify, at instantiation time, the network location of the database to use and connectivity information (username and password) for each component.

4.1.2.4 Widget Creation

As stated in 3.5.1.3, to create a widget from scratch a designer must specify the attributes, callbacks and services the widget will provide along with a mechanism for obtaining data from the widget sensor. However, often a designer can reuse an existing widget and subclass it to customize it and obtain useful functionality with minimal effort required. For example, our original Location Widget used an iButton reader as a sensor with the attributes 'location', 'time' and 'id' and a single 'update' callback. It had no services and obtained data from the reader using a provided software development kit. This widget required applications to take the iButton id the widget returned and explicitly call an interpreter to obtain a user name. We created a new Location Widget by subclassing the original Location Widget and slightly modifying it. Where the original Location Widget reported the 'id' of the iButton docked, the new version also reports the 'user name' of the iButton's owner, which it obtains by calling the interpreter itself. This new Location Widget is easier for applications to use and provides the information at the most likely needed granularity. It only required the addition of a single attribute and a call to the interpreter when the sensor made data available. This is just one example of how new widgets can be easily created from existing widgets, similar to techniques used frequently in GUI programming.

4.1.3 Services

In our discussion of the necessary components in a supporting architecture, we discussed the notion of context services (Section 3.5.4). In our implementation of the Context Toolkit, we chose not to make services first class components, but, rather, to incorporate them into widgets. Widgets are typically responsible for obtaining input from the environment, but they should also be responsible for their counterpart, that of performing actuation of output to the environment, much like GUI *interactors* are. For example, a widget that detects the light level in a room could also be responsible for controlling the light level in that room (via lamps or window blinds). This allows a single component to behave as an interactor responsible for input and output, and to logically represent "light" in that room.

When a widget is created, the widget creator specifies what services the widget will support. The creator can either choose from existing services in the component library or can create additional ones. Services may be synchronous or asynchronous and may take any number of parameters, as specified by the service creator. A generic Service class is provided in the Context Toolkit that handles most of the repetitive tasks required for executing a remote service, such as error checking, parameter marshalling and unmarshalling, and returning the results of a service execution.

An application can query a widget for its list of services using the `queryServices` message, introduced in the previous sub-section. When a synchronous service is requested via the application's BaseObject, the message type is `serviceRequest` with the timing parameter set to `synchronous`. In this case, the service performs some behavior and replies immediately with an execution status and any return data. When an asynchronous service is requested, the application specifies an object that will handle the results of the asynchronous service. This object must implement the `asynchronousServiceHandler` interface, much like an object that handles callback notifications implements the `Handler` interface. The `asynchronousServiceHandler` interface has only one method, `asynchronousServiceHandle()`. The message type used when requesting an asynchronous service is `serviceRequest` with the timing parameter set to `asynchronous`. In this case, the service replies immediately with an execution status but continues to perform the requested service. When the service has completed, the widget returns the service result information to the requesting application, using a `serviceResult` message. The application's BaseObject receives this message, replies with a `serviceResultReply` message indicating successful arrival with no errors, and delivers the service data to the appropriate service handler. It determines which handler to use, using a similar mechanism as with the callback handlers. When the widget receives the `serviceResultReply` message, it removes the application from its list of service handlers.

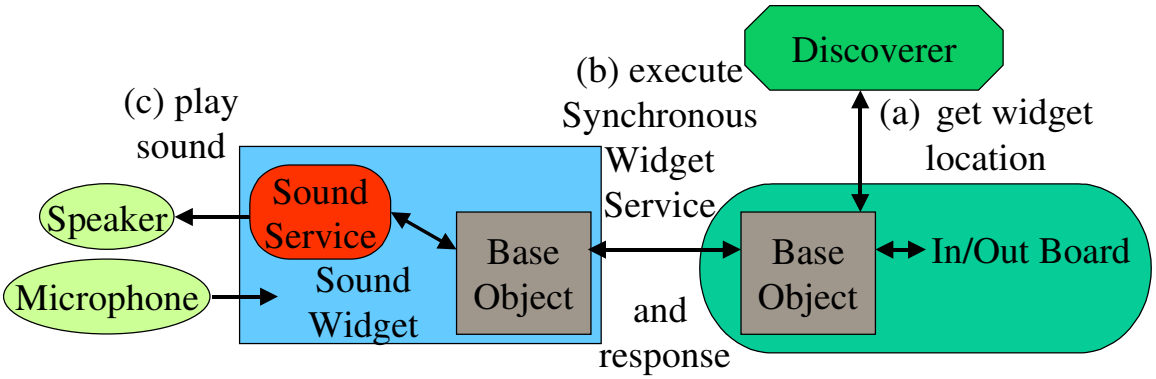


Figure 17: Example of an application executing a synchronous context service.

Figure 17, Figure 18, Figure 19 and Figure 20 show how an application executes a remote context service. Let us first examine the use of a synchronous service in the context of the In/Out Board (Figure 17 and Figure 18). A Sound Widget has a service that can play a sound over a speaker on the computer the widget is running on. (In addition, the Sound Widget is able to detect ambient sound levels in the environment in which it is installed.) The In/Out Board uses the Sound Widget service to play a particular sound file when someone it recognizes enters the building and a different sound file when someone it recognizes leaves the building. The purpose of this service is to provide feedback to users as to when their entrances and exits are recognized. The In/Out Board uses a Discoverer (a) to locate a Sound Widget near the entrance to the building. When users enter or leave the building, the application requests this service to be executed with a particular sound (b) via its BaseObject. The connection between the BaseObject and the widget that houses the service is maintained until the service has executed and returns status information (and any data, if applicable).

```
public DataObject handle(String callback, DataObject data) {
        if (callback.equals("IOBupdate")) {
                // pull out context data from callback and update display as in Figure 15
                …

                // execute synchronous service (b)
                DataObject resultObj = contextConnection.executeSynchronousWidgetService(
                        soundWidgetHost, soundWidgetPort, soundWidgetId,
                        SoundService.PLAY_SOUND, status, new AttributeNameValues());
                String result = resultObj.getDataObject(Service.STATUS).getFirstValue();
                …
        }
}
```

Figure 18: Application code demonstrating use of a synchronous service.

Let us now examine the use of an asynchronous service in the same setting (Figure 19 and Figure 20). An application, say an augmented In/Out Board, upon being notified of a user's presence in the room wants to deliver a message to the user. The message is from another user who wants to know whether the newly arrived user is available for a meeting at 11:30. The message requires a response, thus making it a good candidate to be handled with an asynchronous service. A Display Widget in the room has an asynchronous service that displays messages on a local display, allows a user to respond to the message and then returns the response. The application (a) locates the Display Widget and (b) requests the asynchronous service to be executed (with the message to display) via its BaseObject. When doing so, it must specify a handler that will handle the results of the asynchronous service (similar to a subscription). The widget calls the service

(c), returns status information (an error code) immediately and then removes the connection between the BaseObject and service. The Display Service displays the given message (d) and waits for a response from the user. When the service has a response to return, it sends it to this BaseObject (e) which routes the data to the relevant service handler (f) and calls the method `asynchronousServiceHandle()`.

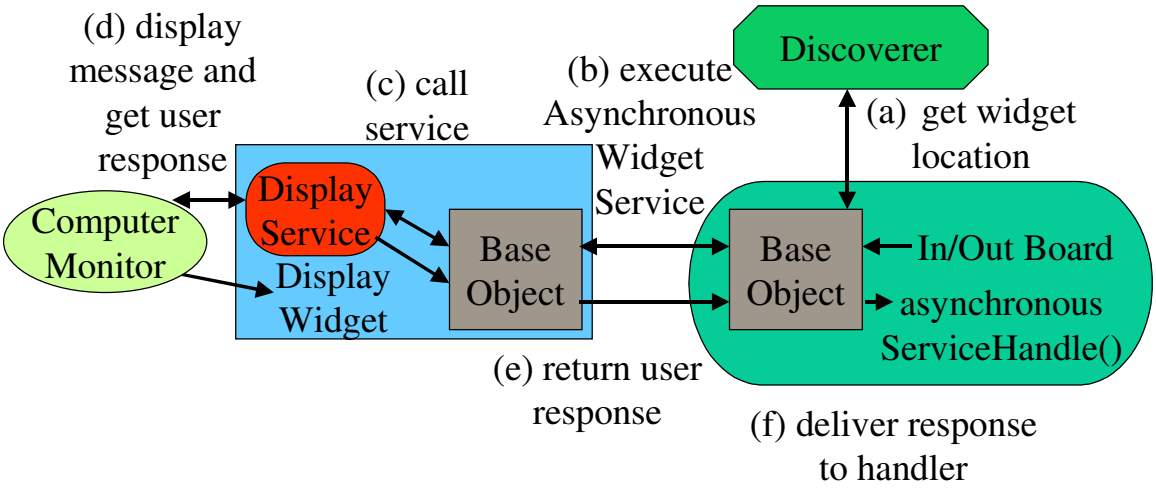


Figure 19: Example of an application executing an asynchronous context service.

```
public class InOutBoard implements Handler, AsynchronousServiceHandler {
        …
        public DataObject handle(String callback, DataObject data) {
                if (callback.equals("IOBupdate")) {
                        // pull out context data from callback and update display as in Figure 15
                        …
                        // construct message and responses to display and store in message variable
                        …
                        // execute asynchronous service, setting the application as the handler for
                        // the results of the service (b)
                        DataObject resultObj = client.executeAsynchronousWidgetService(this,
                                displayWidgetHost, displayWidgetPort, displayWidgetId,
                                DisplayService.DISPLAY, DisplayService.MESSAGE, message,
                                "messageResponse");
                        String result = resultObj.getDataObject(Service.STATUS).getFirstValue();
                        …
                }
                // called when the asynchronous service has data to return (f)
                public DataObject asynchronousServiceHandle(String tag, DataObject data) {
                        if (tag.equals("messageResponse")) {
                                AttributeNameValues anvs = new AttributeNameValues(data);
                                String response =
                                (String)anvs.getAttributeNameValue(RESPONSE).getValue();
                                // do something with response
                                …
        }
}
```

Figure 20: Application code demonstrating use of an asynchronous service.

4.1.4 Discoverer

Applications, and other context components, use a Discoverer to locate context components that are of interest to them, as a form of resource discovery. A Discoverer allows applications to not have to know *a priori* where components are located (in the network sense). It also allows them to more easily adapt to changes in the context-sensing infrastructure, as new components appear and old components disappear. For example, in the In/Out Board application, the application uses the Discoverer to locate any context components that can provide information about people entering or leaving the building it is responsible for. It specifies the attributes it is interested in (location, username, in/out status, and timestamp) and the constant attribute (location = "building X"). The Discoverer uses this information to locate any widgets and aggregators that can return this information, if any. It returns all the information it has about each relevant component, including the hostname or I.P. address of the computer the component is executing on, the port number the component receives communications on and the component's unique identity. The application also subscribes to the Discoverer to be notified about any changes to the components (*e.g.* a component unregisters from the Discoverer) it is using or the appearance of a new component that can also meet the specified conditions. If no component is available, an application-specific decision must be made as to what to do. In some applications, as with the In/Out Board, if no component is available, the application is not useful, so its user interface should be hidden until a component is available. In other applications, the application can continue to do useful work despite a component not being available. But only the application designer can decide this.

Additionally, programmers can use a Discoverer to determine what context can be currently sensed from the environment. It can query the Discoverer for all of the components that are available. We will discuss this idea further in 4.2.3.3, in terms of support for prototyping applications.

The Context Toolkit allows for multiple Discoverers. Each Discoverer is typically responsible for all the components in a physical environment (in much the same way that each Lookup Service in Jini (Sun Microsystems 1999) is responsible for all the services on its computer network). Components are not limited to registering with a single Discoverer and can register with multiple Discoverers, if desired. To connect the information held by multiple distributed Discoverers, each of these Discoverers can register with a higher-level Discoverer, forming a scalable hierarchy or federation of Discoverer objects.

Discoverers can handle the following types of messages or requests:

Discoverer registration

- `addDiscoveredObject`: Register a component with the Discoverer
- `removeDiscoveredObject`: Unregister or remove a component from the Discoverer

Discoverer subscriptions

- `subscribeToDiscoveredObjects`: Subscribe to the Discoverer for discovered objects
- `removeDiscovererSubscriber`: Unsubscribe from the Discoverer

Discoverer queries

- `retrieveKnownDiscoveredObjects`: Query the Discoverer for its discovered objects

When any widget, interpreter or aggregator is started, it registers with a known Discoverer (only the machine hostname or I.P. address of the Discoverer is required). Widgets and aggregators notify a Discoverer of their id, hostname, port, attributes (constant and variable), callbacks, and services. Interpreters notify the Discoverer of their id, hostname, port, attributes that they can interpret and attributes that they can output. To register themselves, components use the `addDiscoveredObject` message type. Discoverers occasionally ping registered components to ensure that these components are still available. If a component is stopped manually, it notifies the Discoverer that it is no longer available to be

used, using the `removeDiscoveredObject` message type. However, in the case of a computer crashing, the component cannot call this method and the Discoverer determines that the component is no longer available when the component does not respond to a specified number of consecutive pings.

Discoverers help make applications more robust to changes in the context-sensing environment. By being notified of components arriving and leaving, applications have the opportunity to adapt to these changes, as described in the In/Out Board example above, rather than just throwing an exception and crashing. Via BaseObject, components and applications can subscribe to be notified (`subscribeToDiscoveredObjects`) when other components register or unregister from the Discoverer. Similar to widget callbacks, components can subscribe with conditions. A component may be interested in only widget or aggregators that have a particular set of attributes, and/or a particular set of constant attributes, and/or services, or interpreters that have a particular subset of incoming and/or outgoing attributes, or components with a particular combination of hostname, port number and/or unique identity. Similarly, components can also unsubscribe (`removeDiscovererSubscriber`) from a Discoverer. When a component registers or unregisters itself with a Discoverer, the Discoverer sends a `discovererUpdate` message to all relevant subscribers, where relevance depends on the subscription conditions, if any. Subscribers to a Discoverer must designate an object, which could be the subscribers themselves, to handle the Discoverer notifications. This object must implement the `Handler` interface, the same interface that is used to handle widget callback notifications. Components can also request a list of all the components discovered so far or a list of components that match a specified set of conditions (`retrieveKnownDiscoveredObjects`).

In Figure 21 and Figure 22, the In/Out Board is used as an example of how applications can interact with the Discoverer. In this example, the application (a) asks the Discoverer for any instances of a Location Widget that have information about Building A. It receives those instances (Location Widgets 1 and 2) and (b) subscribes to them. It also (c) subscribes to the Discoverer to determine when any new instances are available or any existing instance disappears. When a new Location Widget (number 3) (d) is started and becomes available, it (e) registers with the Discoverer. The Discoverer (f) notifies the In/Out Board's BaseObject and this BaseObject (g) routes the discovery information to the application's `handle()` method. The `handle()` method then (h) subscribes the In/Out Board to the newly discovered Location Widget.

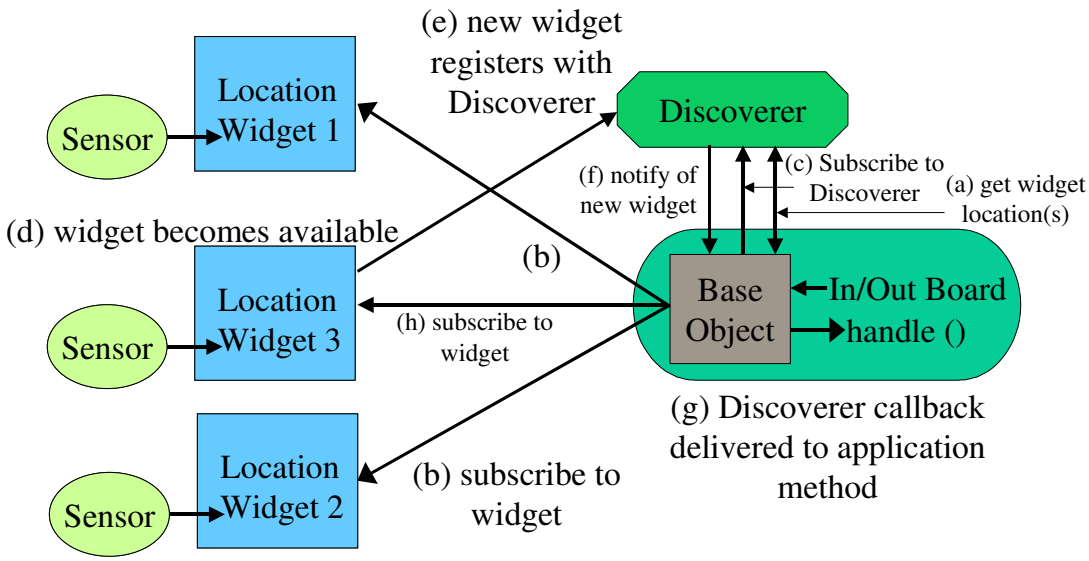


Figure 21: Example of an application interacting with the Discoverer.

```
public class InOutBoard implements Handler {

        // pass in “Building A” as value for location parameter
        public InOutBoard(String location) {
                // create BaseObject to enable communications with infrastructure – receives
                // communications on port number defined by PORT
                BaseObject client = new BaseObject(PORT);

                // code to determine network location (widgetHost and widgetPort) and
                //  identity of relevant widgets (widgetId) (a)
                Attributes attributes = new Attributes();
                attributes.addAttribute(USERNAME);
                attributes.addAttribute(IN_OUT);
                attributes.addAttribute(TIMESTAMP);
                AttributeNameValues constants = new AttributeNameValues();
                constants.addAttributeNameValue(LOCATION, location);
                DataObject discover = client.retrieveDiscoveredObjects(discovererHost,
                        attributes, constants);
                Error discoverError = new Error(discover);
                if (discoverError.getError().equals(Error.NO_ERROR)) {
                        discoveredObjects = new DiscoveredObjects(discover);
                }
                else {
                        discoveredObjects = new DiscoveredObjects();
                }

                // code to subscribe to widget for information about Gregory – repeat for each user
                // Sets this class as the handler for the callback information, subscribes to the widget’s
                // locationUpdate callback, and sets a local callback of IOBupdate (b)
                Conditions conditions = new Condtions();
                conditions.addCondition(USERNAME, Storage.EQUAL, “Gregory”);

                for (int i=0; i< discoveredObjects.numDiscoveredObjects(); i++) {
                        DiscoveredObject object = discoveredObject.getDiscoveredObjectAt(i);
                        String widgetHost = object.getHostname();
                        int widgetPort = object.getPort();
                        int widgetId = object.getId();
                        String widgetCallback = object.getCallback();
                        Error error = client.subscribeTo(this, PORT, “InOutBoard”, widgetHost,
                                widgetPort, widgetId, widgetCallback, “IOBupdate”, conditions,
                                attributes);
                }

                // subscribe to Discoverer for appearance/disappearance of relevant components
                // set the application to be the handler for this information (c)
                Error subError = server.subscribeDiscoveredObjects(this, discovererHost,
                        SUBSCRIBER_ID, DEFAULT_PORT, attributes, constants, "discovery");
                if (!(subError.getError().equals(Error.NO_ERROR))) {
                        System.out.println("Problem subscribing to Discoverer: "+subError.getError());
                }


                …
```

```
        }

        public DataObject handle(String callback, DataObject data) {
                if (callback.equals("IOBupdate")) {
                        …
                }
                // discovery callback (g)
                else if (callback.equals("discovery")) {
                        String action = (String)data.getDataObjectFirstValue(Discoverer.ACTION);
                        if (action.equals(Discoverer.ACTION_ADDITION)) {
                                // subscribe to new component as above (h)
                                …
                        }
                        else { //removal update
                                DiscoveredObject object = new DiscoveredObject(data);
                                Error error = client.unsubscribeFrom(object.getHostname(),
                                        object.getPort(), object.getId());
                                …
                        }
                }
        }
}
```

Figure 22: Application code demonstrating querying and subscription to the Discoverer.

4.1.5 Interpreters

Interpreters are responsible for mapping between context representations and for performing complex inferences on collections of context. For example, an interpreter can convert GPS coordinates to a street location. Or, an interpreter can combine the context available in a conference room to determine whether or not a meeting is occurring. Interpreters are necessary when widgets do not or cannot provide context at the level required by applications. Any component can provide context to an interpreter and have that context interpreted. Interpreters can be run on any available computer, as long as it is accessible to the rest of the context architecture.

A basic Interpreter object that sub-classes from BaseObject is provided in the Context Toolkit. It allows other components, via their instance of BaseObject, to:

- `queryInAttributes`: inspect the context types that an interpreter instance can interpret;
- `queryOutAttributes`: inspect the context types that an interpreter instance can output; and,
- `interpret`: request interpretation.

In the In/Out Board example, we have assumed that there was a widget that could provide the desired in/out information. In the example below, we will break this assumption. The only relevant widget returns a user id, rather than a username, as with the iButton discussed earlier. In order to convert the user id to a username, an interpreter is required. Figure 23 and Figure 24 show how the In/Out Board acquires context from a widget and calls an interpreter. These figures are the same as Figure 14 and Figure 15, only they include an ID to Name Interpreter. The In/Out Board uses the Discoverer (a), not only to locate the Location Widget, but also to locate the ID to Name Interpreter. When the In/Out Board's `handle()` method is called (e), it extracts the user id from the received context, (f) calls the interpreter to convert the user id to a name and then updates its display. Unlike subscriptions, interpretation is a synchronous request, meaning that a connection to the interpreter is maintained until the interpreter returns some result.

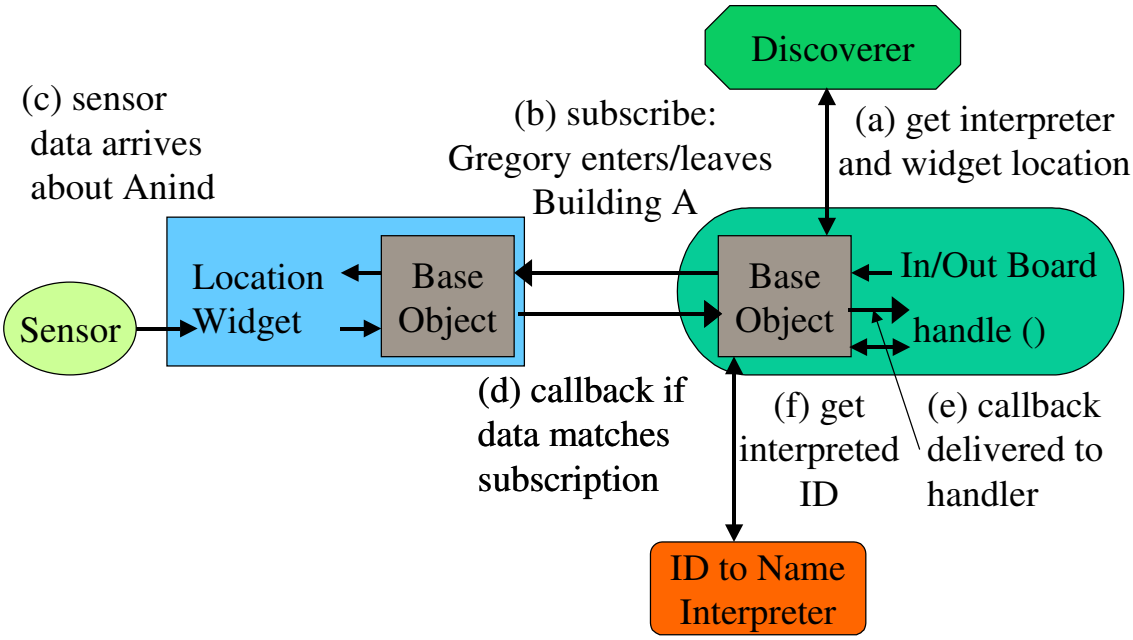


Figure 23: Example of an application subscribing to a context widget and using an interpreter.

```
public class InOutBoard implements Handler {

        …

        public DataObject handle(String callback, DataObject data) {
                if (callback.equals("IOBupdate")) { // (e)
                        AttributeNameValues anvs = new AttributeNameValues(data);
                        String userId = (String)anvs.getAttributeNameValue(USERID).getValue();
                        String status = (String)anvs.getAttributeNameValue(IN_OUT).getValue();
                        Long time = (Long)anvs.getAttributeNameValue(TIMESTAMP).getValue();

                        // call interpreter (f)
                        AttributeNameValues preInterpretAtts = new AttributeNameValues();
                        preInterpretAtts.addAttributeNameValue(WPersonPresence.USERID,userid);
                        DataObject interpret = server.askInterpreter(interpreterHost, interpreterPort,
                                interpreterId, preInterpretAtts);
                        String interpretError = new Error(interpret).getError();
                        AttributeNameValues interpretAtts = null;
                        if (interpretError.equals(Error.NO_ERROR)) {
                                interpretAtts = new AttributeNameValues(interpret);
                                String user = (String)interpretAtts.getAttributeNameValue
                                        (USERNAME).getValue();
                                UpdateDisplay(user,status,time);
                        }

                }
        }
```

Figure 24: Application code demonstrating the use of an interpreter.

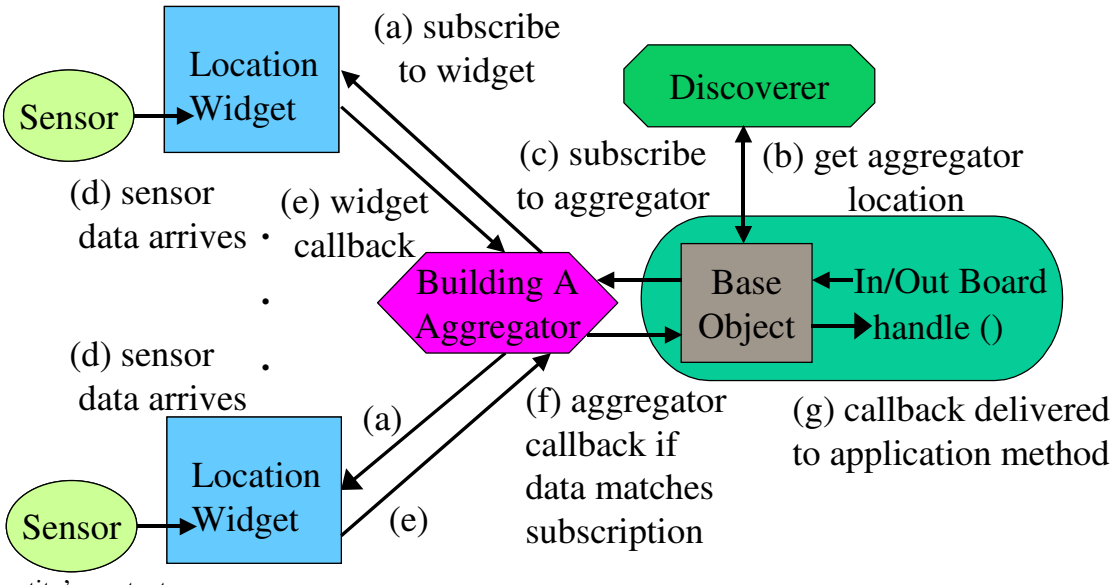
4.1.6 Aggregators

Aggregators are similar in functionality to widgets in that they allow the context they acquire to be subscribed to and queried and they store context. However, they differ in scope. A widget represents all the context from a particular sensor whereas an aggregator is responsible for all the context about a particular entity (person, place or object). Aggregators acquire this context from existing widgets thereby providing an additional degree of separation between the acquisition of context from the use of context. Aggregators can run on any computer, as long as they are accessible to the rest of the context architecture.

A basic Aggregator object that sub-classes from Widget is provided in the Context Toolkit, performing all the tasks of a widget, as specified in 4.1.2. When an aggregator is started, it can either locate appropriate widgets via the Discoverer or it can use a list of widgets provided to it at runtime. The aggregator acts as a mediator between other components and these widgets. When another component wants to communicate with any of these widgets, it can do so via the aggregator. When inspected, the aggregator's attributes, callbacks and services are a union of these widgets' attributes, callbacks and services, respectively.

At runtime, the entity for which the aggregator is responsible is specified. For example, this could be "username = Anind Dey" or "location = Atlanta" or "device name = munch.cc.gatech.edu". This specification is used as a condition when subscribing to relevant widgets, allowing the aggregator to only receive information about its entity.

A final difference between aggregators and widgets is in the way that they locally cache context data. A context widget caches the last data that its sensor provided it. However, an aggregator's local cache contains an entry for each type of context that the aggregator is receiving, which potentially comes from many callbacks. Rather than overwriting the cache when a new callback is received, only the relevant entries are overwritten. In this way, the aggregator's cache maintains the most up-to-date view of its



entity's context.

Figure 25: Example of an application subscribing to a context aggregator.

In Figure 25, the In/Out Board is subscribing to an aggregator, rather than an individual widget. In this case, the Building A Aggregator is aggregating context from a number of widgets (Location Widgets and others) that can provide information about Building A. Figure 25 is very similar to Figure 14, from the application's point of view. The only difference is that the aggregator has subscribed to all the relevant

widgets (a) and is treating each widget as a sensor. The application (b) locates the aggregator and (c) subscribes to it. When the aggregator is notified (e) of a change in data (via a widget callback), it (f) notifies any relevant subscribers, including the In/Out Board's BaseObject. As before, (g) the callback is passed to the appropriate handler. Minimal changes are required to the sample code shown in Figure 15. Only the code used to discover the relevant components changes, since the application is looking for the aggregator that represents Building A as opposed to individual widgets that provide location information about Building A.

4.1.7 Applications

Applications are not actually components within the Context Toolkit, but make use of the components in the toolkit. While widgets, aggregators, interpreters, discoverers sub-class directly from BaseObject, or in the case of aggregators, indirectly, applications are not required to. Our reference implementation for the Context Toolkit was written in Java, which only supports single object inheritance. Therefore, to allow applications to sub-class from other objects, we do not force applications to sub-class from BaseObject or from some Application object, to use the toolkit. Instead, applications typically create an instance of BaseObject that allows them to communicate with the context components.

4.2 Revisiting the Design Process and Architectural Requirements

In CHAPTER 3, we presented an original design process for building context-aware applications as well as a simplified one. We claimed that by providing the correct support, we could allow application designers to move from using the more complex process to using the simplified process. In this section, we will revisit both the design process and the architectural features discussed in CHAPTER 3, and discuss how the Context Toolkit has simplified the designing and evolution of context-aware applications.

4.2.1 Revisiting the Design Process

In Section 3.2, we presented a design process for building context-aware applications and showed how we could reduce the design process by removing accidental activities. We will now describe how the Context Toolkit implementation allows a designer to use the simpler design process.

4.2.1.1 Specification

Specification of what context an application requires is an essential step and cannot be reduced. However, via the Discoverer, an application designer can quickly determine what context is available for use in the chosen runtime environment. Then, the designer can determine the necessary set of queries, subscriptions and interpretations to use to acquire the needed context from the context-sensing infrastructure.

4.2.1.2 Acquisition

If the required context is not available from the infrastructure, a sensor and widget must be added. If the widget does not exist in the component library, then the acquisition of context is a necessary step. Installation of a (real or simulated – see 4.2.3.3) sensor is supported by allowing the sensor (or at least the code connecting the sensor to the context architecture) to be installed on almost any platform and used with a number of programming languages. The code to acquire context is simplified by the provision of a general widget interface to implement. Interpreters allow the conversion of the sensor data into a more usable form. Context storage, communications, and event management mechanisms are already provided for the designer. She only needs to integrate the sensor-specific details, such as obtaining data from the sensor and specifying the attributes and callbacks, with these general mechanisms already available. If a sensor has been used before with such an architecture, the sensor can be reused with minimal effort (the sensor must be physically installed and the widget must be started) on the designer's part.

4.2.1.3 Delivery

The entire activity of delivering context is accidental. Support for distribution and transparent communications mechanisms allows context to be sent between sensors and applications without the designer having to worry about either side.

4.2.1.4 Reception

Reception of context is also an accidental step. General architecture-level features remove this step from the design process. Resource discovery allows applications to easily locate components that provide the context they require. Independent execution of the architectural components combined with resource discovery lets applications use both sensors that were available when the application was started and sensors that were made available after the application was started. The storage of context and the provision of querying and notification mechanisms allow applications to retrieve both current and historical context on a one-time or continuous basis, under the conditions they specify.

4.2.1.5 Action

When the context-aware action is application-specific, providing support for the action is an essential step and must be provided by the application designer and the features discussed in this chapter provide no assistance with this step. However, when the context-aware action is a generic action, not tied to an application, remote context services can be created, if necessary, or existing services can be used.

4.2.2 Revisiting the Architecture Requirements

In CHAPTER 3, we introduced the required features for an architecture that supports the building and execution of context-aware applications:

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

In the following sections, we will briefly discuss how the Context Toolkit addresses each of these features.

4.2.2.1 Context Specification

Applications can specify what context they require to the Discoverer and find the components they require for acquiring and transforming the context. Once the components have been found by an application, the application can create a series of subscriptions and queries to individual widgets and request interpretation from individual interpreters. For example, when the In/Out Board application is started, it uses the Discoverer to locate Location Widgets (and aggregators) that can provide information about occupancy in Building A, widgets that have Display and Sound Services, and ID to Name Interpreters. It first must query the Location widgets to find the current status of each occupant and then subscribe to them to be notified of changes. When it receives context updates, it has the user identity interpreted into a name (via the interpreter), requests the Sound and Display services, and updates its display.

This component abstraction supports being able to acquire single or multiple pieces of context and to acquire filtered or unfiltered context. With our conceptual framework, applications still have to deal with individual components, first locating them and then interacting with them. As well, this abstraction does not support the automatic acquisition of both related and unrelated context (context about multiple entities) and interpreted and un-interpreted context. A higher-level abstraction that builds on top of the component abstraction, which we will discuss in Section 6.3, will address both of these issues.

4.2.2.2 Separation of Concerns and Context Handling

The context widget isolates the details of how context is sensed from those components and applications that simply want to use the sensed context. Users of widgets can essentially treat all widgets in the same manner, with only the data they provide being different. Aggregators build on this notion of abstraction by acting as a repository for a particular entity with the details of how the entity's context was sensed abstracted away.

4.2.2.3 Context Interpretation

Context interpreters support this requirement that there be some mechanism for converting single or multiple pieces of context into higher-level information.

4.2.2.4 Transparent Distributed Communications

BaseObject supports the need for transparent distributed communications. All components (widgets, interpreters and aggregators) and applications use BaseObject instances to communicate with each other. None of these individual components need to know how the distributed communications is being implemented. In the absence of network errors, BaseObject successfully hides the fact that the communications is even distributed.

4.2.2.5 Constant Availability of Context Acquisition

Context widgets, aggregators and interpreters run independently of applications. When applications are started and ended, the context components continue to run as normal. All the context components inherit from BaseObject, which contains a multi-threaded server. With one thread always waiting for incoming communications, the components stay alive forever (barring a crash of the host computer). For widgets and aggregators, if their host crashes, a subscription log allows them to reestablish communications with the components and applications that have subscribed to them. In addition, through resource discovery, applications and components can be notified when other components are available or are no longer available and allows them to find other components that meet their requirements.

4.2.2.6 Context Storage

Context widgets and aggregators automatically store all the context they acquire (although this can be turned off by setting a flag at instantiation time). The storage occurs transparently, so individual widget and aggregator creators do not have to worry about supporting this feature themselves.

4.2.2.7 Resource Discovery

The Discoverer component supports the resource discovery requirement. All context components register themselves and their capabilities with the Discoverer automatically. Applications can flexibly query and subscribe to the Discoverer for components that interest them.

4.2.3 Non-functional Requirements

In addition to the architecture requirements we specified in CHAPTER 3, we specified three other requirements for the Context Toolkit: the ability to be used in heterogeneous environments, the ability to support alternative implementations of architecture features and the ability to prototype applications in the absence of real sensors.

4.2.3.1 Support for Heterogeneous Environments

The need for distribution has a clear implication on architecture design stemming from the heterogeneity of computing platforms and programming languages that could be used to both collect and use context. Unlike an application built with GUI widgets, the programming languages used by the application to communicate with context widgets and used by the context widgets themselves, may not be the same. The architecture should support interoperability of context widgets and applications on heterogeneous platforms.

A feature that is useful for integrating the context architecture into existing applications is the use of programming language and computer platform-independent mechanisms. By making few architectural requirements on the support from programming languages and platforms, we can more easily re-implement the architecture on any desired platform in any desired programming language. This allows designers to work on the platforms and with languages that are convenient for them. This aids in building new applications as well as in adding context to previously non-context-aware applications. There is an additional reason for being able to implement the architecture on multiple heterogeneous platforms. Because sensors will typically be distributed in the environment running on remote platforms, there is the real possibility that the platforms will not be all of the same type. If the architecture is available on multiple platforms, we will easily be able to support the interoperability of components running on different platforms, written in different programming languages.

Most of the components and applications have been written in Java, which allows us to meet our goal of supporting the context architecture on multiple platforms. There is support for Java on most common hardware platforms and operating systems available today. We have demonstrated that the architecture executes under UNIX, Windows 95/98/NT, Windows CE, and Macintosh OS. We have a few components and applications that were written in C++, Frontier, and PERL and have demonstrated runtime interoperability of these with other components and applications written in Java (APPENDIX B).

4.2.3.2 Support for Alternative Implementations

We have provided default implementations for each of the features we have discussed so far. To support maximum flexibility, designers should be able to easily replace the default implementations or policies with their own. This eases the integration of the architecture into existing applications and eases maintainability, by allowing designers to use implementations that they are familiar with and supporting consistency across the application. For example, if an application uses a particular message format for communicating with distributed objects, the designer may find that application development is easier if the default context architecture message format is replaced with this distributed object message format. Supporting alternative implementations also allows designers to easily explore the impact of various design decisions on performance and usability.

While all the default implementation of features can be replaced at compile time, we wanted to support easy replacement by allowing it to occur at instantiation time. In our Java version of the toolkit, by providing a pointer (the name of the class) to the alternative implementation, the instantiated component can automatically replace the default implementation and use the alternative implementation.

For example, in BaseObject, the communications scheme can be replaced. As discussed earlier, it supports communications using HTTP and XML for the sending and receiving of messages. Although using this combination supports the operation of the toolkit on heterogeneous platforms and programming languages, the combination is quite slow. XML is a verbose language with lots of overhead required to send data and HTTP is a slow protocol. For this reason, BaseObject supports the use of other communications protocols and data languages. When a designer wants an object to use a different protocol, SMTP (Simple Mail Transfer Protocol (Postel 1982)) for example, she creates two objects that "speak" SMTP, one for outgoing communications (that adds the SMTP around the outgoing message) and one for incoming communications (that strips the SMTP from the incoming message). These objects must implement the `CommunicationsClient` and `CommunicationsServer` interfaces, respectively. She then passes the names of these objects to the BaseObject at the time of instantiation. The BaseObject uses these objects rather than the default HTTP communications objects. The current implementation of the Context Toolkit requires that all of the components use the same communications protocol. A component using HTTP for its protocol cannot communicate with a component using SMTP for its protocol. In addition, the interfaces currently limit the use of communications protocols to those that are TCP/IP-based.

In a similar fashion, a designer can replace XML with use his own data-encoding scheme. For example, if a designer had a custom data format (or HTML, for example) that he wanted to for sending messages, he has to provide two objects that understand the new data format, one for outgoing messages (that encodes the DataObject message in the custom format) and one for incoming messages (that decodes the custom format into a DataObject message). These objects must implement the `EncoderInterface` interface and the `DecoderInterface` interface, respectively. The `EncoderInterface` has one method, `encodeData`, and the `DecoderInterface` has one method as well, `decodeData`.

XML and HTTP were chosen for the default implementation because they support lightweight integration of distributed components and allow us to meet our requirement of architecture support on heterogeneous platforms with multiple programming languages. XML is simply ASCII text and can be used on any platform or with any programming language that supports text parsing. HTTP requires TCP/IP and is ubiquitous in terms of platforms and programming languages that support it. Other alternatives to XML and HTTP include CORBA (Object Management Group 2000) and Java RMI (Sun Microsystems 2000a). Both were deemed too heavyweight, requiring additional components (an ORB or an RMI daemon) and in the case of RMI, would have forced us to use a specific programming language – Java. The combination of XML over HTTP is becoming quite popular, being used in Hewlett-Packard's ChaiServer/eSpeak (Hewlett Packard 2000) and Microsoft's Simple Object Access Protocol (SOAP) (Box, Ehnebuske *et al.* 2000).

Widgets use a replaceable storage mechanism. By default, the mechanism for persistent storage uses JDBC to communicate with mySQL. A widget designer not wanting to use the default mechanism can provide an object that allows the storage and retrieval of information from some persistent storage, whether that be a different SQL database, flat-file storage or some other mechanism. At run time, the designer gives the name of the object class to the widget, allowing the new storage mechanism to be used. This object must implement the `Storage` interface, which contains methods to store and retrieve data, to specify what attributes need to be stored, and to flush data from the local cache to persistent storage.

Finally, the resource discovery protocol used by the Discoverer can also be replaced. It currently uses a simple, custom protocol, but can be replaced by a more sophisticated protocol if desired by the toolkit user at runtime. The protocol must be embodied in an object that implements the `Discover` interface. The interface contains methods to register and unregister components, allow querying of the discovered components and allow notifications about changes to the discovered components. Potential replacements for the custom resource discovery protocol are Jini (Sun Microsystems 1999) (although it is a Java-only solution), Salutation (Salutation Consortium 2000), the Service Location Protocol (SVRLOC Working Group of the IETF 1999) and Universal Plug and Play (Universal Plug and Play Forum 2000). Using a standard discovery protocol may allow for some interoperability with components not written for the Context Toolkit.

4.2.3.3 Support for Prototyping Applications

A third non-functional requirement of the Context Toolkit is that it support the prototyping of context-aware applications, even in the absence of available sensors. Often when dreaming up interesting applications, we are thwarted because the applications require sensors that either have not yet been invented or are not available for use. We can use a Discoverer to determine what context is available from the environment. If the desired context is not available, we either have to drop the new application idea or determine how to build the application without having the necessary sensors. To this end, the toolkit supports the use of software sensors in the form of GUIs that collect input explicitly from the user software sensors.

For example, Figure 26 shows a GUI that is used to provide context to a Location Widget. When a user clicks on a person's name, that person's in/out status is toggled. Because this Location Widget has the same

interface as other Location Widgets, an application can use any of these Location Widgets interchangeably, regardless of the sensor used, without requiring any change to the application code.

Figure 26: Example of a GUI used as a software sensor.

As a more complex example, an application may need to know where a user is, what his mood is and what his activity is. If an adequate positioning system is not available then a GUI sensor can be used. The interface would display a number of locations and allow the user to choose one in order to indicate his current location. The user input generates context that can be captured and delivered by the widget wrapped around the sensor. There is no sensor currently available to determine what a user's mood is and sensors for determining activity are quite limited. Again, a GUI can be created that allows a user to explicitly specify his mood and activity. By treating the GUIs as sensors and wrapping them with widgets, the application can be prototyped and tested using a Wizard-of-Oz methodology. In addition, when suitable real sensors become available, they can be wrapped with widgets and incorporated into the system without requiring any change to the application.

4.2.4 Design Decisions

When building the Context Toolkit, we had to make a number of design decisions for implementing various features and functionality. In this section we will discuss some of these design decisions, in particular, looking at the tradeoffs between a distributed system and a centralized system.

4.2.4.1 View of the world

There are two main ways that context-aware application developers can view the world of context. The first is to view the world of context is a set of components that map to real world objects and locations (Schilit 1995; Dey, Salber *et al.* 1999; Minar, Gray *et al.* 2000). Widgets represent sensors and actuators and aggregators represent people, places, and objects. The second way to view the world of context is as a blackboard that can be filled with different types of context (Cohen, Cheyer *et al.* 1994; Davies, Wade *et al.* 1997; Sun Microsystems 2000b; Winograd 2001). Components can place context onto the board and applications can take information off of the board.

The conceptual framework and the Context Toolkit use the real world view as opposed to the blackboard view. There are advantages and disadvantages to both types of representations. The real world view allows application developers to think about and design their applications in terms of the sensors and individual entities (people, places and things) that exist in the real world. This is a natural and straightforward way to view the context world. However, it forces developers to deal with components on an individual basis, requiring a more procedural style of programming, and makes it difficult to perform an analysis of context across components. Procedural programming requires that a designer encode what an application is supposed to do to achieve a particular result. When communicating with individual components, a configuration problem arises where applications and components must have knowledge of where other components reside.

The blackboard view allows application developers to design their applications in terms of a large context store, dealing only with the context they require and not having to worry about individual components. Since each component and application deals with the blackboard, the configuration problem seen in the real world view is greatly simplified, with each component only having to know the location of the blackboard. However, the blackboard representation tends to be more centralized and, therefore, suffers from having a single point of failure. This blackboard view supports a more declarative style of programming and allows for easier analysis of context that originated from multiple components. Declarative programming allows a designer to describe what result is desired and leave it to the infrastructure to determine how to achieve that result. However, the blackboard view does not accurately represent the real world and may be more difficult for designers to use. In addition, blackboards do not support storage of context, but can only represent the current state of a system.

We chose the real world view for implementing the Context Toolkit for its features of being more natural in representing the real world and of not suffering from a single point of failure. In addition, the component representation has proven to be easy to extend and build on top of. As we will show in CHAPTER 6, the widget metaphor makes it quite easy to control access to a user's context, for the reasons of privacy and security (6.1). In particular, we will see the disadvantage of blackboards being essentially context-free and not knowing anything about the components that write data to them or take data from them or the semantics of the data itself.

As well, we will show how the widget metaphor makes it easier to handle and resolve uncertainty in context data (6.2). Finally, we will show that the component representation can be extended to gain the advantages of the blackboard representation (6.3), including easier configuration (for the application developer), and support for the higher-level procedural-style programming and analysis of context across individual components, without the single point of failure concern. This allows the programmer to choose the level of abstraction they want to use for programming (individual components or the world as a whole) and the style of programming (procedural or declarative).

4.2.4.2 Data Storage

As described earlier (4.1.2.3), when a widget is instantiated for the first time, it automatically creates a table in a database to use for storing the context it acquires from its sensor. Aggregators do the same to store context they acquire from their widgets. Once again a tradeoff was made between a centralized storage mechanism and a distributed storage mechanism. The centralized mechanism suffers from the single point of failure problem and can suffer from performance problems if the context store is being read from and written to frequently, but is easier to set up and maintain. The distributed mechanism does not have the single point of failure problem and is less likely to suffer performance degradation (since the reads and writes will be distributed among many components). It is, however, harder to set up and maintain. We chose the distributed mechanism for our implementation. The database that a widget uses can vary from widget to widget and can be configured at runtime. The widget storage mechanism encapsulates the table setup, so individual widget creators do not have to deal with this issue.

4.2.4.3 Context Delivery

When a widget acquires new context from its sensor, it sends the context to its subscribers individually, (unicast) and not to all the subscribers at once (broadcast or multicast). The tradeoff in this decision was between saving bandwidth and supporting simplicity. The broadcast/multicast communications scheme is both simpler and more efficient (*i.e.* more scalable) from the widget's perspective. However, it sends unnecessary data to subscribers, requiring them to perform any desired filtering to find the relevant context, if any, from a callback message, making the subscriber development more complex. The unicast communications scheme is slightly more complex and is not as efficient, but it only sends data that subscribers have explicitly asked for. It allows subscribers to subscribe to callbacks with particular conditions and list of attributes they are interested in. This makes application (subscriber) development

simpler because the filtering is done by the widget and saves network bandwidth. We chose the unicast method because it supported our main objective of making it easier for designers to build applications.

4.2.4.4 Context Reception

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 traditional GUI event queue mechanism does not apply well to context. Rather than placing received events sequentially in a queue, all components listen for context messages constantly. When a message arrives, it is handed over to a new thread that processes the message (much like web servers do). Thus, a component is always listening for messages, and messages are handled in parallel.

Handling context messages in this way allows context to be handled in parallel, but suffers from being resource intensive. 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 0.01 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.

4.2.4.5 Programming Language Support

As described in our discussion of non-functional requirements, we made a decision to avoid programming language-specific capabilities to allow cross-platform and cross-language interoperability. Once again, we made a tradeoff. Here it was between supporting a large developer population and supporting performance and scalability. The Context Toolkit is one implementation of the conceptual framework described in CHAPTER 3. While other future implementations can be made more efficient and scalable, our goal with this implementation is to explore the space of context-aware computing. By choosing interoperability over efficiency, we have opened up the developer community to those that do not or cannot use a single programming language or computing platform, enabling a more thorough exploration by ourselves and others.

4.3 Summary of the Context Toolkit

In this chapter, we presented an implementation of the conceptual framework described in CHAPTER 3, called the Context Toolkit. The Context Toolkit consists of five main components, BaseObject that is responsible for supporting communications between context components and applications, context widgets, context interpreters, context aggregators and discoverers. Examples of the communications between components and applications were provided to demonstrate how the Context Toolkit is used to build both context components and context-aware applications. In addition to describing how the Context Toolkit implements the requirements of a framework that supports context-aware applications, we also described three non-functional requirements of the toolkit: support for developing and executing in heterogeneous environments, support for allowing alternative implementations of architectural features and support for prototyping applications when physical sensors are not available. Finally we discussed the tradeoffs made in implementing the conceptual framework.

CHAPTER 5

BUILDING APPLICATIONS WITH THE CONTEXT TOOLKIT

In this chapter, we will discuss a number of different applications that have been designed and implemented with the Context Toolkit. Our goal is to demonstrate that our architecture can support applications that do not suffer from the problems we introduced earlier:

- a general lack of context-aware applications
- a lack of variety of sensors;
- a lack of variety of context types; and,
- an inability to evolve applications.

The applications we will present are:

- In/Out Board and Context-Aware Mailing List: demonstrates use of simple widget that is reusable by multiple applications and demonstrates how applications can evolve to use different sensors;
- DUMMBO: demonstrates evolution of an application from being non-context-aware to being context-aware;
- Intercom: demonstrates a complex application that uses a variety of context and components; and,
- Conference Assistant: demonstrates a complex application that uses a large variety of context and sensors.

Together these applications cover a large portion of the design space of context-aware applications, using time, identity, location, and activity as context and perform the context-aware features of presenting context information, automatically executing a service based on context, and tagging captured information with context to promote easier retrieval.

5.1 In/Out Board and Context-Aware Mailing List: Reuse of a Simple Widget and Evolution to Use Different Sensors

In this section, we will 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. They will demonstrate how applications can be evolved to use different sensing mechanisms and multiple sensors. We will describe how each application is used and how each was built.

5.1.1 In/Out Board

5.1.1.1 Application Description

The first application we will present is the In/Out Board that we have used as an example throughout this thesis. The In/Out Board built with the Context Toolkit is located at the entrance to our research lab (Salber, Dey *et al.* 1999a). It not only displays the in/out status (green/red dot) of building occupants, but also displays the time when the occupants last entered/left the building. It is our longest running application, serving its users for over two years. A Web-based version of the In/Out Board (Figure 27a) 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 I.P. 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 27b). 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. All of these applications support the context-aware feature of presenting information and services to a user.

(a)

(b)

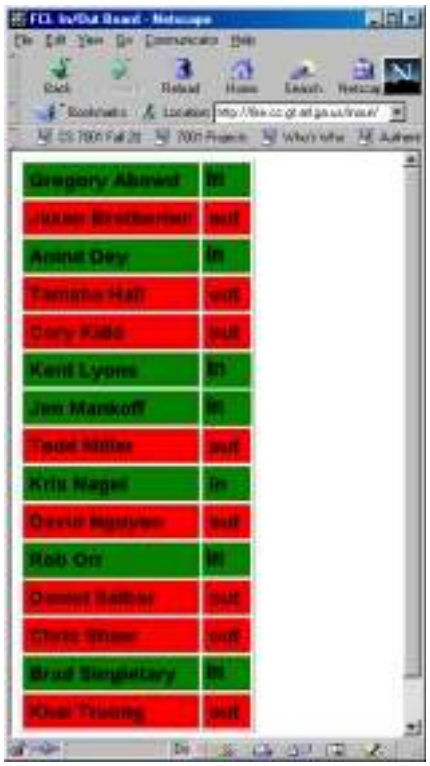


Figure 27: Standard (a) and web-based (b) versions of the In/Out Board application.

5.1.1.2 Application Design

The following figure (Figure 28) shows the components that are used to build both the standard application and the Web-based application. A Presence widget detects users as they enter and leave our research area. We have two implementations of the Presence widget. One version (more commonly used), detects user presence when users explicitly dock their Java iButton into a Blue Dot Receptor (an iButton reader). The widget uses this docking event to toggle the user's status, from 'in' to 'out' or 'out' to 'in'. The second implementation uses the radio frequency-based PinPoint 3D-iD indoor location system to automatically detect user presence (PinPoint 1999). 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 'out'. 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. All of these components are executing on the same machine for convenience purposes.

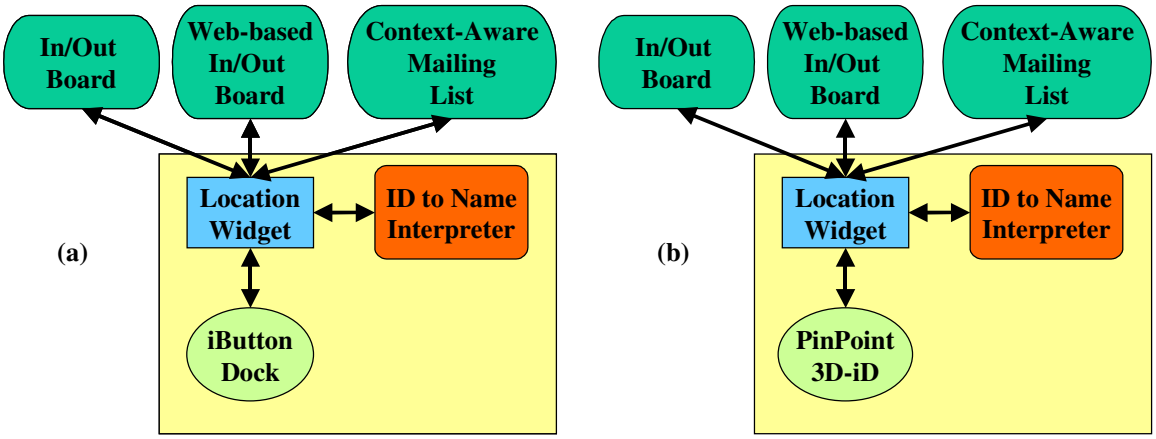


Figure 28: Architecture diagrams for the In/Out Board and Context-Aware Mailing List applications, using (a) iButtons and (b) PinPoint 3D-iD for location sensors.

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 (CGI) 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 I.P. address is used to control whether time information is included in the generated Web page.

5.1.2 Context-Aware Mailing List

5.1.2.1 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 sub-group on the list. The problem is compounded when the sub-group's constituents are dynamic. The Context-Aware Mailing List is an application that delivers e-mail messages to members of a research group (Dey, Salber *et al.* 1999). It differs from standard mailing lists by delivering messages only to members of the research group who are currently in the building. For example, a user is about to go out for lunch and wants to know if anyone wants to join her. She could send a message to everyone in her

research group, but the message would not be appropriate for those not in the building. They would see it at some later time when the message would no longer be applicable. Instead she sends it to the Context-Aware Mailing List, which delivers it to only those users who are in the building. This application supports the context-aware feature of automatically executing a service.

5.1.2.2 Application Design

Figure 28 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 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.

5.1.3 Toolkit Support

The In/Out Board and the Context-Aware Mailing List are examples of simple applications that demonstrate reuse of a context widget and interpreter. Both applications required the same type of context, presence of an individual in defined location. 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 (Salber, Dey *et al.* 1999a). 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. This ability allows us to easily evolve our context-aware systems, in terms of the sensing technologies, and to prototype with a variety of sensors.

We built these applications using the context component abstraction that the toolkit supports, in terms of widgets and interpreters. 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 insulated from run-time widget instantiations and crashes.

5.2 DUMMBO: Evolution an Application to Use Context

In this section, we will present DUMMBO (Dynamic Ubiquitous Mobile Meeting Board) (Brotherton, Abowd *et al.* 1999; Salber, Dey *et al.* 1999a) that demonstrates how we can evolve an application that does not use context to one that uses context. It leverages off context not to offer new features, but to make its existing features perform more effectively and appropriately.

5.2.1 Application Description

DUMMBO was an already existing system that we chose to augment. It is an instrumented digitizing whiteboard that supports the capture and access of informal and spontaneous meetings (Figure 29).

Captured meetings consist of the ink written to and erased from the whiteboard as well as the recorded audio discussion. After the meeting, a participant can access the captured notes and audio by indicating the time and date of the meeting.

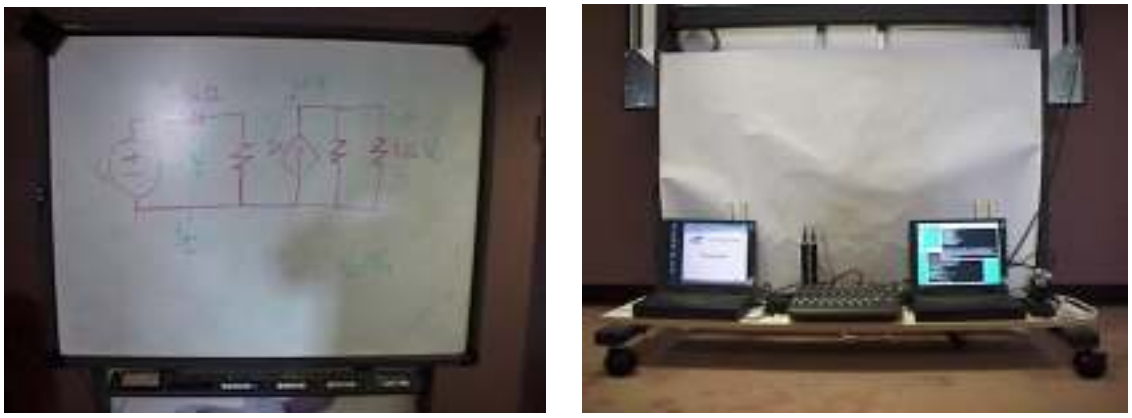


Figure 29: DUMMBO: Dynamic Ubiquitous Mobile Meeting BOard. (a) Front-view of DUMMBO. (b) Rear-view of DUMMBO. The computational power of the whiteboard is hidden under the board behind a curtain.

In the initial version of DUMMBO, 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 capture of this discussion, 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 30 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 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.

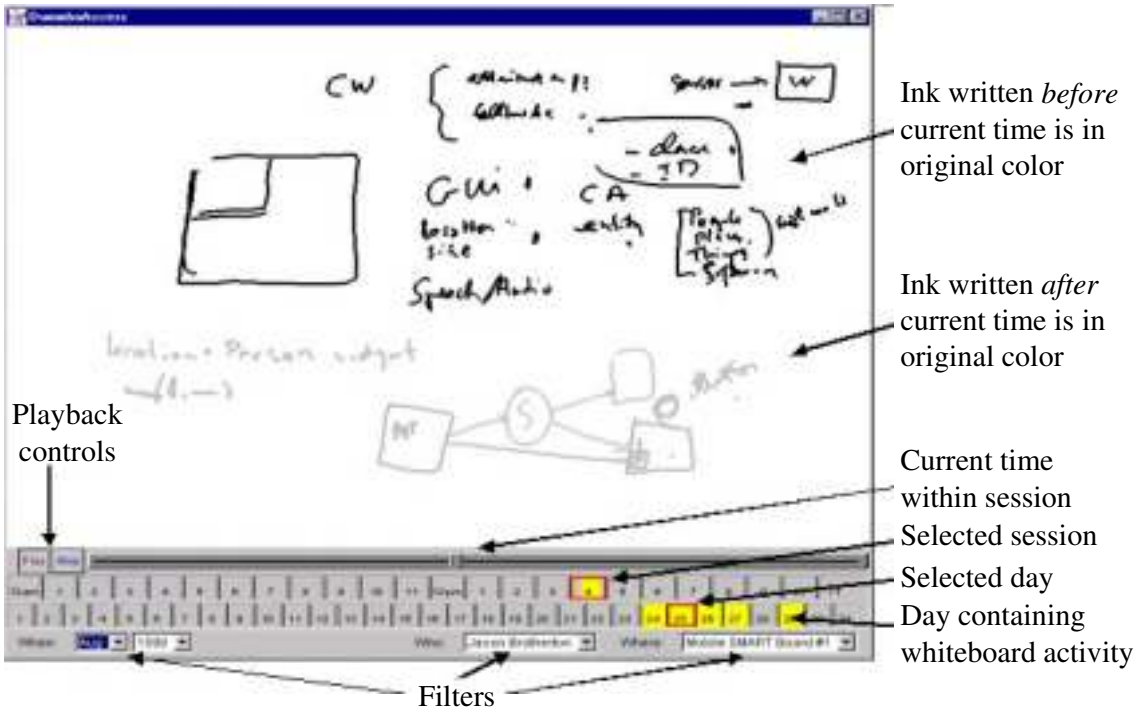


Figure 30: DUMMBO access interface. The user selects filter values corresponding to when, who, and where. DUMMBO then displays all days containing whiteboard activity. Selecting a day will highlight all the sessions recording in that day. Playback controls allow for live playback of the meeting.

5.2.2 Application Design

The context architecture used to support the DUMMBO application is shown in Figure 31. It used a single Location widget (using iButtons and docks) to detect which users were around the mobile whiteboard. It used multiple Location Widgets (also using iButtons) to determine where the whiteboard was located in our lab – there was one widget for each location where DUMMBO could be moved. This widget and the application were running on the same machine. The interpreter and the other widgets were running on a number of other computers. 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 events about 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 about a change in the whiteboard position, it tags the meeting being captured with the new meeting location information.

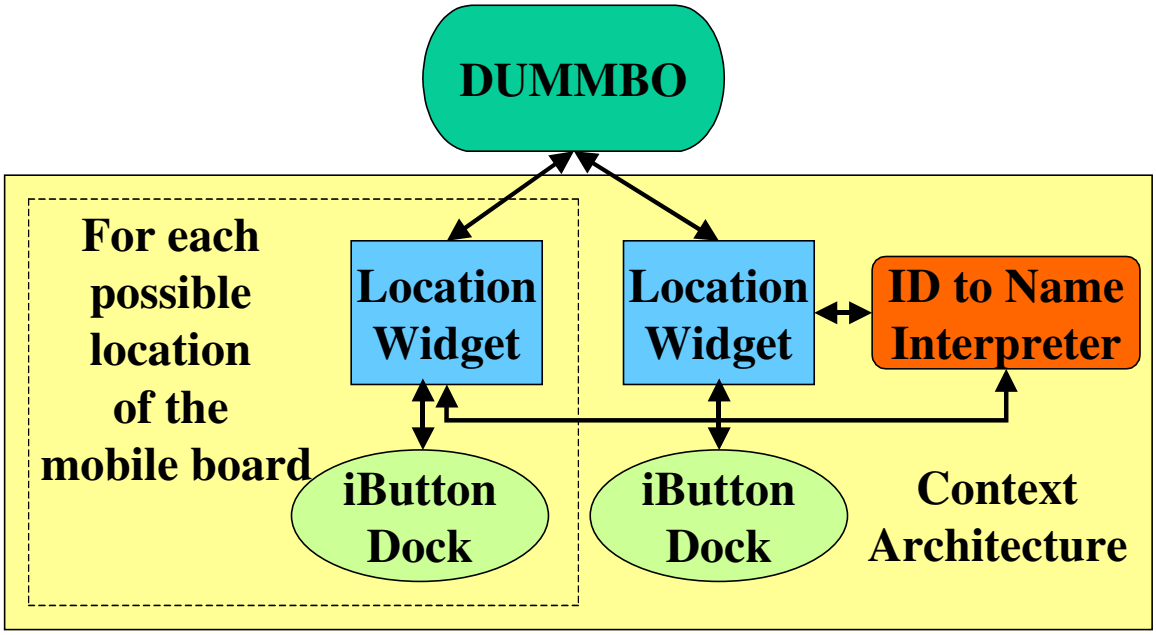


Figure 31: Context architecture for the DUMMBO application.

5.2.3 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 12400 lines of Java code. The Context Toolkit has made it easier for programmers to use context without having to write large amounts of code, by letting them leverage off of both its infrastructure and its library of reusable components.

Unlike the previous applications, this one used multiple widgets. All of the widgets were of the same type, providing location 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 addition, this application used context to enhance its existing features, that of determining when to start recording activity and of making it easier to retrieve captured meetings.

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. There is a tradeoff to be made here. To reduce the number of lines of code, the programmer must create an aggregator for the whiteboard, if one did not already exist, and execute an additional context component.

5.3 Intercom: Complex Application that Uses a Variety of Context and Components

The Intercom application is an application that demonstrates the use of context beyond simple location (Kidd, O'Connell *et al.* 2000), including co-location, what a users say and whether users are involved in conversations. It does use location information (like most context-aware applications) and demonstrates the use of multiple sensors to acquire it. It also demonstrates the use of aggregators and services. It was developed by members of our research group and did not involve anyone involved in the design of the Context Toolkit.

5.3.1 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/her location *(e.g.* broadcasting is often used) and the lack of knowledge of the recipient'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.

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. If Sam were involved in a conversation with someone else in the house, the initiator would be notified of the situation and could try to initiate the conversation later.

5.3.2 Application Design

The context architecture used to support the Intercom application is shown in Figure 32. 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 in combination with a Location Widget that used the WEST WIND RF-based beaconing system (Lyons, Kidd *et al.* 2000), were 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 recipient) to use the intercom. Upon receiving a request, the application queries the Person Aggregator for the caller and recipient to determine their respective locations and, in the case of a conversation request, queries the Room Aggregator for the recipient to determine if anyone else is in the room. The application queries the Audio Widget to determine if the user is in a conversation with anyone else and uses 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 recipient). If the caller approves the connection (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 recipient 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 in order to avoid multiple conversations in a single 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. Each of the widgets, the interpreter and application are running on a separate machine. The aggregators are all running on a single machine.

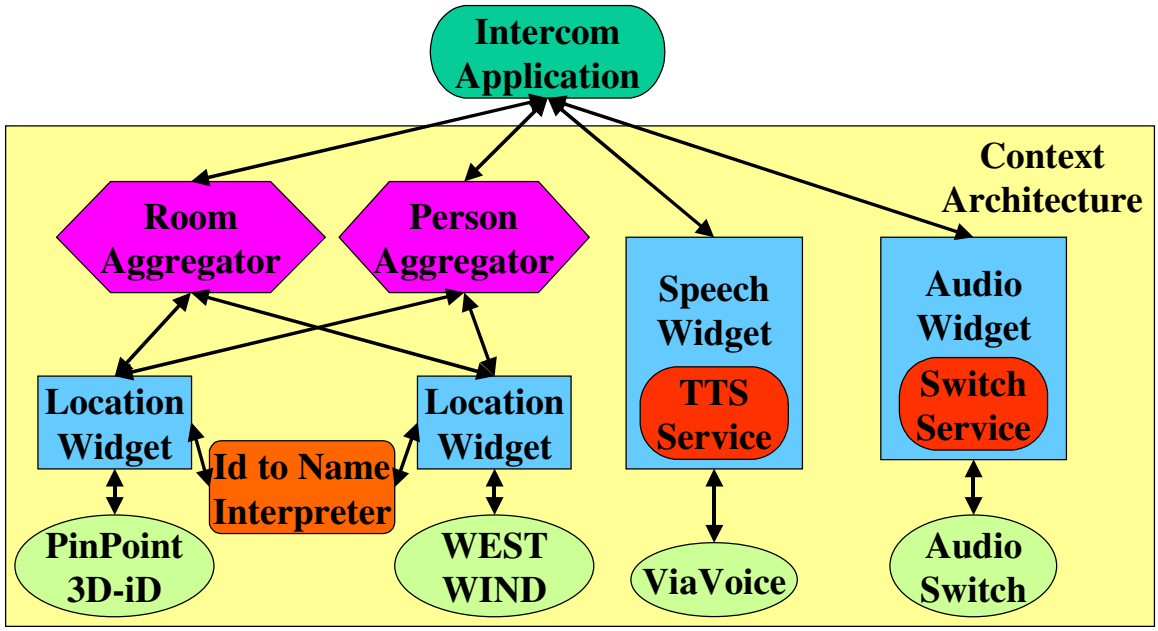


Figure 32: Context architecture for the Intercom application.

5.3.3 Toolkit Support

The Intercom application is a fairly sophisticated context-aware application that, like DUMMBO, was built by other members of our research group that did not include any developers of the Context Toolkit. It displays context to the user when indicating the presence of another individual (co-location) in the recipient's room and when indicating that the recipient is already in a conversation, 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 Intercom application, like the previously discussed applications, used different sensors to acquire context. The difference in this application is that the sensors were used simultaneously. This did not require any change to the application as context acquisition from the Location widgets was mediated through the Person and Location Aggregators (for each person and room, respectively). The developers of this application were able to reuse the PinPoint-based Location Widget, Speech Widget and the Room and Person Aggregators from previously developed applications, leaving only the Audio Widget, the WEST WIND-based Location Widget and the application to be developed. Reuse of these components made the design and implementation of the Intercom system much easier.

5.4 Conference Assistant: Complex Application that Uses a Large Variety of Context and Sensors

The Conference Assistant is the most complex application that we have built with the Context Toolkit (Dey, Futakawa *et al.* 1999). It uses a large variety of context including user location, user interests and colleagues, the notes that users take, interest level of users in their activity, time, and activity in the space around the user. A separate sensor senses each type of context, thus the application uses a large variety of sensors as well. This application spans the entire range of context types we identified in 1.1.2 and the entire range of context-aware features we identified in 1.2.3.

5.4.1 Application Description

We identified a number of common activities that conference attendees perform during a conference, including identifying presentations of interest to them, keeping track of colleagues, taking and retrieving notes and meeting people that share their interests. The Conference Assistant application currently supports all but the last conference activity. The following scenario demonstrates how the Conference Assistant is used. Note that the Conference Assistant was a prototype application that was never deployed at an actual conference. We believe that our implementation will support the number of software components required by a small conference with multiple tracks, however hardware and logistical requirements stopped us from actually deploying the application. All of the features and interfaces described below have been implemented and tested in a very scaled-down simulation of a conference.

A user is attending a conference. When she arrives at the conference, she registers, providing her contact information (mailing address, phone number, and email address), a list of research interests, and a list of colleagues who are also attending the conference. In return, she receives a copy of the conference proceedings and a Personal Digital Assistant (PDA). The application running on the PDA, the Conference Assistant, automatically displays a copy of the conference schedule, showing the multiple tracks of the conference, including both paper tracks and demonstration tracks. On the schedule (Figure 33), certain papers and demonstrations are highlighted (light gray) to indicate that they may be of particular interest to the user.

Figure 33: Screenshot of the augmented schedule, with suggested papers and demos highlighted (light-colored boxes) in the three (horizontal) tracks.

The user takes the advice of the application and walks towards 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 of the presentation are being recorded. This impacts the user's behavior, taking fewer or greater notes depending on the extent of the recording available. The presenter is using a combination of PowerPoint and Web pages for his presentation. A thumbnail of the current slide or Web page is displayed on the PDA. The Conference Assistant allows the user to create notes of her own to "attach" to the current slide or Web page (Figure 34). As the presentation proceeds, the application displays updated information for the user. The user takes notes on the presented slides and Web pages using the Conference Assistant. The presentation ends and the presenter opens the floor for questions. The user has a question about the presenter's tenth slide. She uses the application to control the presenter's display, bringing up the tenth slide, allowing everyone in the room to view the slide in question. She uses the displayed slide as a reference and asks her question. She adds her notes on the answer to her previous notes on this slide.

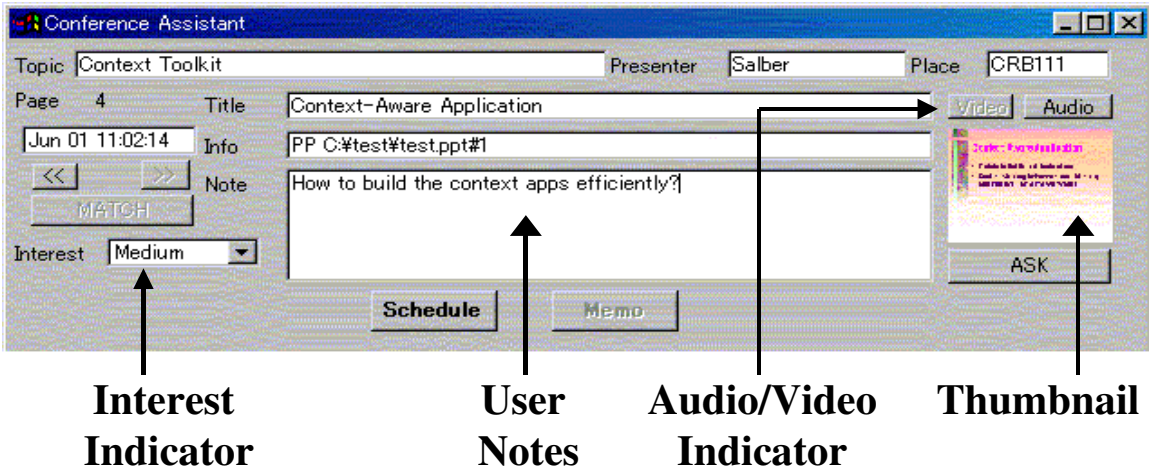


Figure 34: Screenshot of the Conference Assistant note-taking interface.

Figure 35: Screenshot of the partial schedule showing the location and interest level of colleagues. Symbols indicate interest level.

After the presentation, the user looks back at the conference schedule display and notices that the Conference Assistant has suggested a demonstration to see based on her interests. She walks to the room where the demonstrations are being held. As she walks past demonstrations in search of the one she is interested in, the application displays the name of each demonstrator and the corresponding demonstration. She arrives at the demonstration she is interested in. The application displays any PowerPoint slides or Web pages that the demonstrator uses during the demonstration. The demonstration turns out not to be relevant to the user and she indicates her level of interest to the application. She looks at the conference schedule and notices that her colleagues are in other presentations (Figure 35). A colleague has indicated a high level

of interest in a particular presentation, so she decides to leave the current demonstration and to attend this presentation. The user continues to use the Conference Assistant throughout the conference for taking notes on both demonstrations and paper presentations.

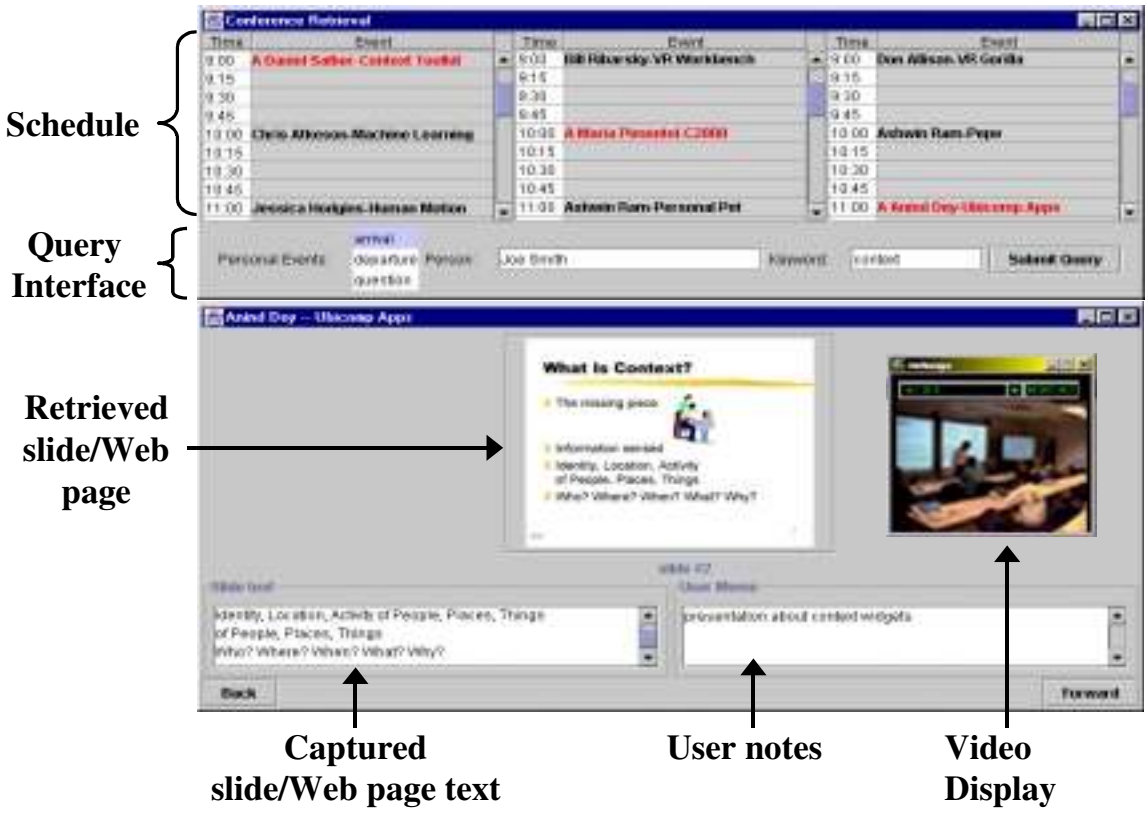


Figure 36: Screenshots of the retrieval application: query interface and timeline annotated with events (a) and captured slideshow and recorded audio/video (b).

She returns home after the conference and wants to retrieve some information about a particular presentation. The user executes a retrieval application provided by the conference. The application shows her a timeline of the conference schedule with the presentation and demonstration tracks (Figure 36a). The application uses a feature known as context-based retrieval (Lamming and Flynn 1994). 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 36a), the user can view (Figure 36b) the slide or Web page presented at the time of the event, audio and/or video recorded during the presentation of the slide, and any personal notes she may have taken on the presented information. She can then continue to view the current presentation, moving back and forth between the presented slides and Web pages.

In a similar fashion, a presenter can use a third application to retrieve information about his/her presentation. The application displays a timeline of the presentation, populated with events about 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 interface is similar to that shown in Figure 36. The presenter can 'relive' the presentation, by playing back the audio and/or video, and moving between presentation slides and Web pages.

The Conference Assistant is the most complex context-aware application we have built. It uses a wide variety of sensors and a wide variety of context, including real-time and historical context. This application supports all three types of context-aware features: presenting context information, automatically executing a service, and tagging of context to information for later retrieval.

5.4.2 Application Design

The application features in the above scenario presented have all been implemented. The Conference Assistant makes use of a wide range of context. In this section, we discuss the types of context used, both in real time during a conference and after the conference and how they were used to provide benefits to the user. We will then describe the context architecture that the application leveraged from.

When the user is attending the conference, the application first uses information about what is being presented at the conference and her personal interests to determine what presentations might be of particular interest to her. The application uses her location, the activity (presentation of a Web page or slide) in that location and the presentation details (presenter, presentation title, whether audio/video is being recorded) to determine what information to present to her. The text from the slides is being saved for the user, allowing her to concentrate on what is being said rather than spending time copying down the slides. The context of the presentation (presentation activity has concluded, and the number and title of the slide in question) facilitates the user's asking of a question. The context is used to control the presenter's display, changing to a particular slide for which the user had a question.

The list of colleagues provided during registration allows the application to present other relevant information to the user. This includes both the locations of colleagues and their interest levels in the presentations they are currently viewing. This information is used for two purposes during a conference. First, knowing where other colleagues are helps an attendee decide which presentations to see herself. For example, if there are two interesting presentations occurring simultaneously, knowing that a colleague is attending one of the presentations and can provide information about it later, a user can choose to attend the other presentation. Second, as described in the user scenario, when a user is attending a presentation that is not relevant or interesting to her, she can use the context of her colleagues to decide which presentation to move to. This is a form of social or collaborative information filtering (Shardanand and Maes 1995).

After the conference, the retrieval application uses the conference context to retrieve information about the conference. The context includes public context such as the time when presentations started and stopped, whether audio/video was captured at each presentation, the names of the presenters, the presentations and the rooms in which the presentations occurred and any keywords the presentations mentioned. It also includes the user's personal context such as the times at which she entered and exited a room, the rooms themselves, when she asked a question and what presentation and slide or Web page the question was about. The application also uses the context of other people, including their presence at particular presentations and questions they asked, if any. The user can use any of this context information to retrieve the appropriate slide or Web page and any recorded audio/video associated with the context.

After the conference, a presenter can also use the conference context to obtain information relevant to his/her presentation. The presenter can obtain information about who was present for the presentation, the times at which each slide or Web page was visited, who asked questions and about which slides. Using this information, along with the text captured from each slide and any audio/video recorded, the presenter can playback the entire presentation and question session.

The Conference Assistant was built using the context components listed in Table 3. Figure 37 presents a snapshot of the architecture when a user is attending a conference. For multiple users and presentations, the user and presentation architecture segments are replicated appropriately.

Table 3: Architecture components and responsibilities in the Conference Assistant.

A = Aggregators, W = Widgets, I = Interpreters

Component	Responsibility
Registration (W)	Acquires contact info, interests, and colleagues
Memo (W)	Acquires user's notes and relevant presentation info
Recommend (I)	Locates interesting presentations
User (A)	Aggregates all information about user
Question (W)	Acquires audience questions and relevant presentation info
Location (W)	Acquires arrivals/departures of users
Content (W)	Monitors PowerPoint or Web page presentation, capturing content
Recording (W)	Detects whether audio/video is recorded
Presentation (A)	All information about a presentation

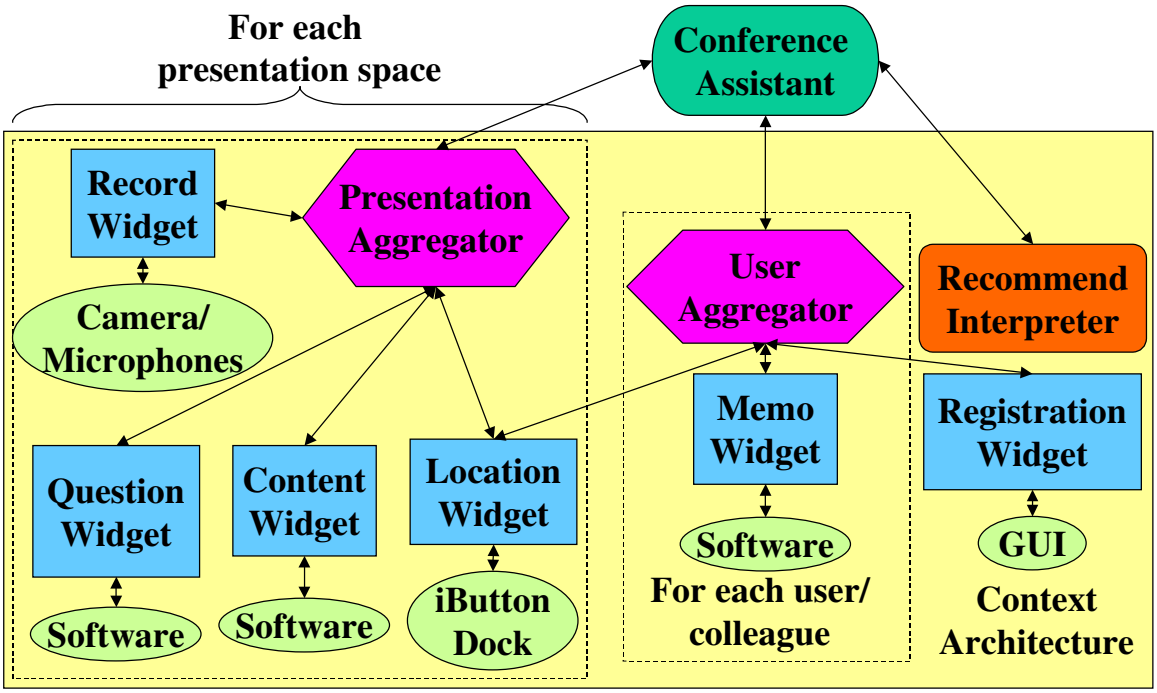


Figure 37: Context architecture for the Conference Assistant application during the conference.

During registration, a User Aggregator is created for the user. It is responsible for aggregating all the context information about the user and acts as the application's interface to the user's personal context information. It subscribes to information about the user from the public Registration Widget, the user's Memo Widget and the Location Widget in each presentation space. The Memo Widget captures the user's notes and also any relevant context (relevant slide, time, and presenter identity). The Registration Widget uses a GUI to collect a user's registration information. The Memo widget also uses a software sensor that captures the notes that a user enters in the Conference Assistant application. For this application, we used both iButton- and PinPoint-based Location Widgets.

There is a Presentation Aggregator for each physical location where presentations/demos are occurring. A Presentation Aggregator is responsible for aggregating all the context information about the local presentation and acts as the application's interface to the public presentation information. It subscribes to the widgets in the local environment, including the Content Widget, Location Widget, Recording Widget

and Question Widget. The Content Widget uses a software sensor that captures what is displayed in a PowerPoint presentation and in an Internet Explorer Web browser. The Recording Widget uses information from a camera and microphones in the presentation space to determine whether audio or video is being captured. The Question widget is also a software widget that captures what slide (if applicable) a user's question is about, from their Conference Assistant application.

When an audience member asks a question using the Conference Assistant, the Question Widget captures the context (relevant slide, location, time, and audience member identity) and notifies the local Presentation Aggregator of the event. The aggregator stores the information and also uses it to access a service provided by the Content Widget, displaying the slide or Web page relevant to the question.

The Conference Assistant does not communicate with any widget directly, but instead communicates only with the user's User Aggregator, the User Aggregators belonging to each colleague and the local Presentation Aggregator. It subscribes to the user's User Aggregator for changes in location and interests. It subscribes to the colleagues' User Aggregators for changes in location and interest level. It also subscribes to the local Presentation Aggregator for changes in a presentation slide or Web page when the user enters a presentation space and unsubscribes when the user leaves. It also sends its user's interests to the Recommend Interpreter to convert them to a list of presentations that the user may be interested in. The Interpreter uses text matching of the interests against the title and abstract of each presentation to perform the interpretation.

Only the Memo Widget runs on the user's handheld device. The Registration Widget and associated interpreter run on the same machine. The User Aggregators are all executing on the same machine for convenience, but can run anywhere, including on the user's device. The Presentation Aggregator and its associated widgets run on any number of machines in each presentation space. Only the Content Widget needs to be run on a particular computer, the computer being used for the presentation.

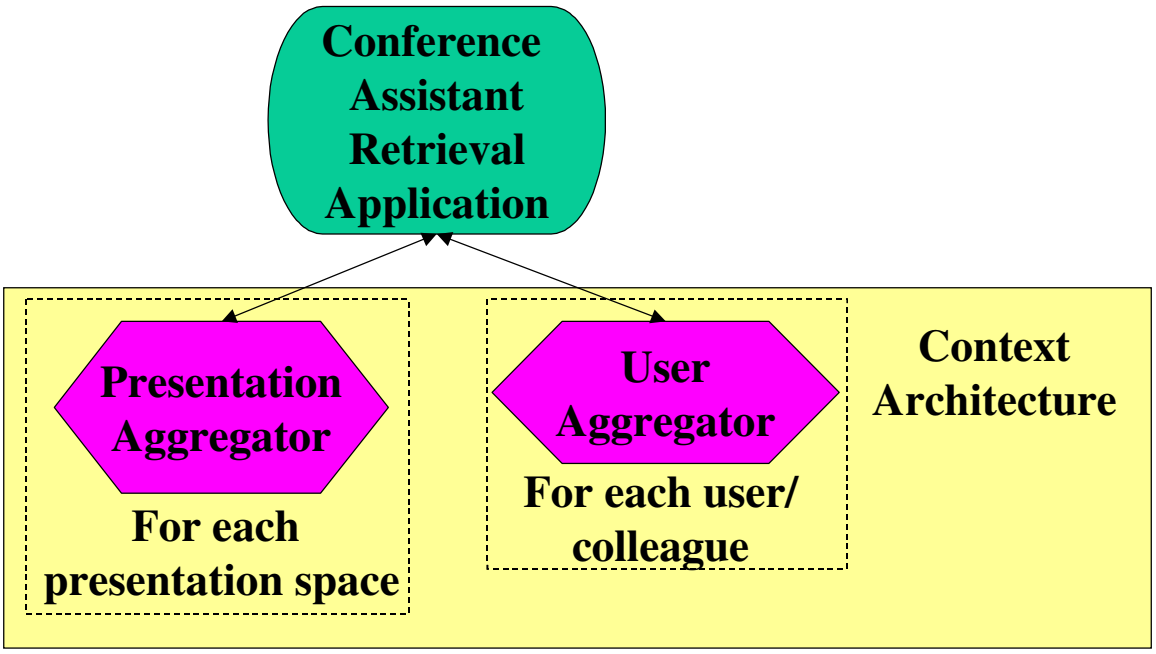


Figure 38: Context architecture for the Conference Assistant retrieval application.

In the conference attendee's retrieval application, all the necessary information has been stored in the user's User Aggregator and the public Presentation Aggregators. The architecture for this application (Figure 38) is much simpler, with the retrieval application only communicating with the user's User Aggregator and each Presentation Aggregator. As shown in Figure 36, the application allows the user to retrieve slides (and the entire presentation including any audio/video) using context via a query interface. If personal context is used as the index into the conference information, the application polls the User Aggregator for the times and location at which a particular event occurred (user entered or left a location, or asked a question). This information can then be used to poll the correct Presentation Aggregator for the related presentation information. If public context is used as the index, the application polls all the Presentation Aggregators for the times at which a particular event occurred (use of a keyword, presence or question by a certain person). As in the previous case, this information is then used to poll the relevant Presentation Aggregators for the related presentation information.

In the presenter's retrieval application, all the necessary information has been stored in the public Presentation Aggregator used during the relevant presentation. The architecture for this application is simple as well, with the retrieval application only communicating with the relevant Presentation Aggregator. As shown in Figure 36, the application allows the user to replay the entire presentation and question session, or view particular points in the presentation using context-based retrieval.

5.4.3 Toolkit Support

The Conference Assistant, as mentioned earlier, is our most complex context-aware application. It supports interaction between a single user and the environment and between multiple users. Looking at the variety of context it uses (location, time, identity, activity) and the variety of context-aware services it provides (presentation of context information, automatic execution of services, and tagging of context to information for later retrieval), we see that it completely spans our categorization of both context and context-aware services. This application would have been extremely difficult to build if we did not have the underlying support of the Context Toolkit. We have yet to find another application that spans this feature space.

As stated earlier, the Conference Assistant was a prototype application that was never deployed at an actual conference. We believe that the Context Toolkit implementation would support the number of software components required by a small conference with multiple tracks. It was the hardware requirements that stopped us from actually deploying the application. The need for a large number of wireless handheld devices, a wireless LAN, and an indoor positioning system makes deployment difficult.

This application used multiple aggregators, one for each user (conference attendee) and one for each presentation space (paper presentation or demonstration presentation). Figure 37 demonstrates the advantage of using aggregators quite well. Each Presentation aggregator collects context from 4 widgets. Each User aggregator collects context from the Memo and Registration widgets plus a Location widget for each presentation space. Assuming 10 presentation spaces (3 presentation rooms and 7 demonstration spaces), each User aggregator is responsible for 12 widgets. Without the aggregators, the application would need to communicate with 42 widgets, obviously increasing the complexity. With the aggregators and assuming 3 colleagues, the application just needs to communicate with 14 aggregators (10 Presentation and 4 User), although it would only be communicating with one of the Presentation aggregators at any one time.

To determine a user's level of interest in a particular presentation, we had the user enter it explicitly using a GUI widget. Currently, we do not have a sensor or the inferencing ability to determine automatically when the user is interested or bored, although there is some interesting research going on in the area of affective computing that we could, one day, leverage from (Picard 1997). The menu that users used to indicate their level of interest is essentially a software sensor that allowed us to build the application and a useful feature in the absence of any available physical sensor.

5.5 Summary of Application Development

In this chapter, we have described a number of context-aware applications that we have built with the Context Toolkit. The In/Out Board and Context-Aware Mailing List are examples of applications that reuse the same infrastructure and context components and that have been made resilient to changes in the context-sensing infrastructure. DUMMBO is an example of an existing application that had context-aware behavior added to it. The Intercom is an example of a complex application that uses a variety of context and components to facilitate communications within a home. Both DUMMBO and the Intercom demonstrate that other researchers have been able to use the Context Toolkit to build context-aware applications. The Conference Assistant is an example of a complex application that uses a variety of context to aid attendees of a conference.

The toolkit facilitated the building of these applications by providing abstractions or context components that make designing easier. The context components automatically provide the low-level and common details that normally have to be supported on an individual basis by the application programmer. The toolkit allows application developers to concentrate on the details of the application and not worry about the details of acquiring and transforming context.
But, using the context component abstraction, application designers still have to worry about some unnecessary details, due in part to the emphasis on components. Referring back to our earlier discussion of the context specification mechanism (3.3.1), we see that there were four requirements. These were the ability to specify:

- unfiltered context vs. filtered context;
- single piece of context vs. multiple pieces of context;
- if multiple, related context vs. unrelated context; and,
- uninterpreted context vs. interpreted context.

The context component abstraction only supports the first two requirements. Filtering of context data is supported through the use of the combination of callbacks, attribute subsets and conditions. This is available in all context widgets and aggregators. If a single piece of context is required, it can be obtained from either a widget or an aggregator. If multiple related pieces are required, then it is best obtained from an aggregator, which is responsible for collecting multiple related pieces of context. But, if the multiple pieces of context are not related (*i.e.* not about a common entity), then an application must request the context from multiple components. If the application wants interpreted context, it first must acquire the context (from potentially multiple components) and then request interpretation from a context interpreter.

While the context components encapsulate their information into logical objects, they force application designers to take extra steps in acquiring the context they require. First, in order to find the components they require, designers must use the resource discovery mechanism. Now that the designer knows what components are available, she must determine what combination of queries and subscriptions is required to obtain the context required by the application. Finally, with individual queries and subscriptions, the application designer must assemble all the acquired context information together and try to determine when an interesting application situation, which potentially stretches over multiple context widgets and aggregators, has been achieved. These three steps are accidental. In the next chapter, we will introduce an extension to the Context Toolkit called the situation abstraction that addresses these issues.

CHAPTER 6

USING THE CONTEXT TOOLKIT AS A RESEARCH TESTBED FOR CONTEXT-AWARE COMPUTING

One of the goals of the Context Toolkit is to make it easier to build context-aware applications, so that we could start to investigate some of the harder issues in context-aware computing, essentially using the toolkit as a research testbed. With context-aware applications being so difficult to build in the past, it was hard to examine challenging issues like controlling access to context, dealing with quality of service concerns and handling inaccurate context. To examine these types of issues, the toolkit must be flexible and extensible. In this chapter, we will describe two preliminary extensions to the Context Toolkit to deal with mediating access to context data to address controlling access to context for privacy and security concerns and to investigate a method for dealing with inaccurate context data. We will then describe an extension to the Context Toolkit that supports a higher-level programming abstraction than context components and makes it even easier to build and evolve context-aware applications.

6.1 Investigation of Controlling Access to Context

In this section, we will present a preliminary investigation on the issue of controlling access to context in the Context Toolkit.

6.1.1 Motivation

One of the biggest social concerns with context-aware applications is that in order to provide useful services to their users, they must sense information about the users. There is a tradeoff that has to be made. The more information that a user is willing to have known about herself, the more sophisticated that the context-aware services can be. If the services are seen as valuable, then users may be more willing to give up a greater level of privacy. Naturally, this tradeoff makes people uncomfortable because they do not know what is being sensed about them or how it is being used (Want, Schilit *et al.* 1995). Giving people control over the information that is sensed about them is one way in which we can alleviate some of the discomfort (Lau, Etzioni *et al.* 1999).

6.1.2 Controlling Access to User Information

To give users true control over their own information, they have to have ownership of the information. Once they have ownership, they can specify who else should be given access to that information and under what circumstances, just as they would with any other personal belongings. Then, when a third party requests a certain piece of information about a user, these rules can be used to decide whether the third party has been granted access to the information. There are four components that are used to implement the access control mechanism. These are Mediated Widgets, Owner Permission, modified BaseObject and the Authenticator. Users specify the access policies or rules and the infrastructure uses these components to support them. We will not discuss these four components.

Users are given ownership by augmenting the context acquisition process. The basic Widget object is sub-classed to create a MediatedWidget object. This new object is exactly the same as the basic Widget object except that it requires a widget developer to specify who owns the information being sensed. For example,

an aggregator that represents a user specifies that the user owns all of its information. A widget that provides information about the presence of people in a certain location specifies that the person about whom the update is about owns each presence update. This sets up simple rules of ownership.

Next, an additional component, called the Owner Permission component, allows owners of information to specify when others should be able to access their information. A user creates a set of situations to indicate what information third parties can access and when. For example, a user may allow access to their location information to colleagues, only between 9:00 am and 5:00 pm on weekdays, but may allow close friends and family access to that information at all times. The Owner Permission component is the keeper of all the specified situations for all the owners of information. When asked to determine permission, it receives a requesting identity and the request (containing the information owner). It steps through each of the situations specified by the relevant owner looking for one that indicates whether the requestor should be granted access to the requested information. Unless a situation is found that grants access, the requestor is denied.

The third piece required is a modified mechanism for requesting context. The BaseObject component provides a number of different methods for acquiring context from the available infrastructure including querying for context, requesting notification about context changes, and asking for historical context. These mechanisms have been modified so that applications and other components using them are required to identify themselves. Without identifying themselves, the infrastructure could not determine what information they should have access to.

A fourth component, called the Authenticator, is used in combination with the modified BaseObject to ensure that the requesting component is who it claims to be. As well, the Authenticator is used to authenticate that a context component (that is being queried for information) is who it claims to be. For example, an application that is acting on behalf of its user David, wants to know where Anind is. When it requests this information from a relevant context widget via its instance of BaseObject, it specifies that it represents David (using standard public-private key infrastructure). The context widget takes the information provided and ensures that the requesting entity is who it claims to be. If authentication is successful, the context widget sends the David identity and the request for information to the Owner Permission component, to ensure that David is permitted to have access to this information about Anind. If the authentication is unsuccessful or David is not permitted to have the requested information, the widget sends back an error message to the requesting application. However, if David is permitted access, then the widget collects the requested context and delivers it back to the application, along with a copy of its identity (and public key). The application uses the information to authenticate that the component returning information is indeed the widget that the application wanted information from.

The Context Toolkit made it quite easy to implement this support for controlling access to user information. The separation of concerns provided by the toolkit allowed us to keep the details on how access control was implemented separate from the application. To make use of the modified architecture, applications only need to provide an identity to the modified BaseObject along with their usual request for information. As foreshadowed in 4.2.4.1, the widget abstraction (and, in general the component representation and the real world view of context) was essential for making the controlled access of user context easy to support. Context widgets are responsible for encapsulating the context that a sensor provides. They are appropriate for assigning ownership because they are the source of the context and the small variety of context that each encapsulates makes it relatively simple to assign ownership. Aggregators are also useful components as they are quite appropriate for holding owner permission information, since they already encapsulate information about users.

Finally, when responding to a request for information, widgets already possess the capability to filter what information they return. For example, a MediatedWidget may provide scheduling information to interested applications. If a user is not authorized to access all the details of the schedule, but only when a user is free, the MediatedWidget is capable of providing the correct level of detail. It is able to filter data according to

the user's access privileges (using the filtering ability it already has) and, more importantly, it understands the semantics of what a schedule is and what the differing levels of detail represent. If a blackboard model were used, a widget (or similar component) could place the scheduling information it has on the blackboard. When the same user wants to access this information, the blackboard is not capable of understanding scheduling semantics (by default), so it would not know how to convert or filter the available scheduling information to the information that the user is allowed to see. Either the blackboard must be given the ability to understand these semantics or the each application must have this ability. Neither situation is desirable. In the first case, the blackboard would need to understand the semantics of all types of context which is quite unrealistic. In the second case, applications would need this ability. In addition, giving applications the ability to control access to information is greatly undesirable.

The MediatedWidget, modified BaseObject and a simplified version of the Owner Permission component have been implemented in this investigation so far. The implemented Owner Permission component contained two types of rules, one set for anonymous or unidentified users and another for identified users. This is an ongoing effort and is the subject of another student's PhD research.

6.1.3 Application: Accessing a User's Schedule Information

An application called the Dynamic Door Display was implemented by other members of our research group with this extension to the Context Toolkit (Nguyen, Tullio *et al.* 2000). A networked display (a 40 x 4 character LCD or a color display from a Hewlett-Packard 620LX handheld computer) is placed on an office door (Figure 39). The display initially displays the names or pictures of the occupants of the office. By selecting an occupant's name, the current known location of the user along with some information about his schedule for the current day (Figure 40). If the user does not identify herself, she is only able to see the times when the user has something scheduled. If the user does identify herself, using an iButton, she is shown a detailed version of the schedule with times and event details. She can also leave a voice message for the occupant. This application is similar to the Context-Aware Office Assistant (Yan and Selker 2000), in that it allows a visitor to interact with a occupant's schedule, but additionally uses context to determine how much information should be provided to the visitor.

Figure 39: Photographs of the Dynamic Door Display prototypes.

Figure 40: Screenshot of the Dynamic Door Display

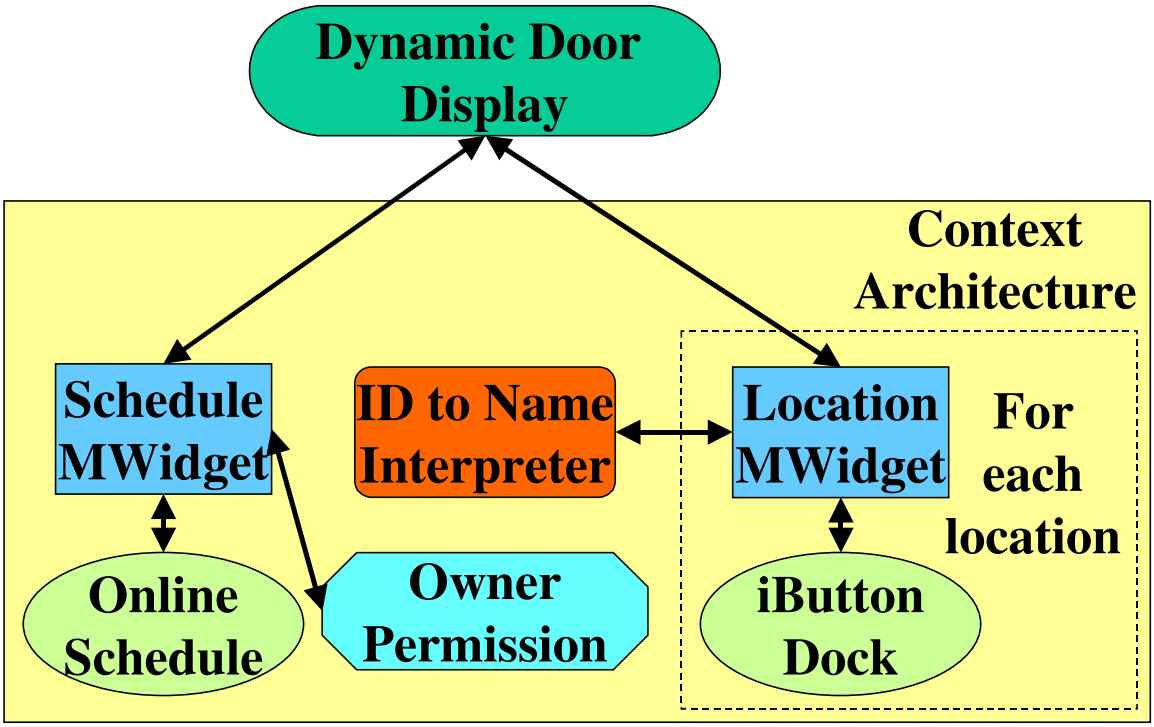


Figure 41: Architecture for the Dynamic Door Display application.

The architecture for the application is shown in Figure 41. The Dynamic Door Display subscribes to Location Widgets so it will know where occupants of the office are at all times. This includes a Location Widget that is connected to the display to allow users to identify themselves. The Location Widgets use the iButton sensors to acquire location information. The application also queries and subscribes to a Schedule Widget that acquires the schedules for each of the occupants from an online calendar. When a user interacts with the Door Display, the application displays the location of the office occupants, using the information from the Location Widgets. If the user requests additional schedule information about a particular user but has not identified herself, the application queries the Schedule Widget with an anonymous identity. If the user has identified herself, the application uses her identity for the query. The Schedule Widget contacts the User Permissions component with the user's identity and the requested information about a particular office

occupant's schedule. The User Permission component returns with a list of information that the user is allowed access to. If the user's identity is anonymous, then the component returns with a subset of the requested attributes (username of the occupant, starttime, endtime) instead of the complete requested list (username, starttime, endtime, location, description). However, if the user identified herself, then the entire list is returned. The Schedule Widget uses the returned list to filter the information that it sends back to the application.

6.1.4 Discussion

Our investigation into mediating access to context information is in its early stages. While the architecture for dealing with controlling access to context seems sound, there are a number of interesting issues still left to address. At the heart of this modified architecture is a rule-based system. Users who have information sensed about them are required to create rules that specify who else can have this information and in what circumstances. While defaults can be provided, no set of defaults can ever hope to be satisfactory for a group of users. The creation of appropriate rules can be quite complex and difficult to write. As well, it is often difficult to write rules that will cover all the circumstances that may arise. A rule-based mechanism that requires users to do so much work may not be appropriate in this work.

Controlling access to information based on the requestor's identity is a form of traditional role-based access control (Sandhu, Coyne *et al.* 1996) that uses subject roles. By extending the rules that users create to include other context beyond the requestor's identity, we can support a more flexible and generalized role-based access control mechanism that uses not only subject roles, but also object and environment roles and combinations of all three (Covington, Moyer *et al.* 2000; Covington, Long *et al.* 2001). Object roles correspond to objects in the environment and their properties. Environment roles correspond to time, locations and their properties. For example, a user may create a rule stating that her colleagues can only access information about her schedule between the hours of nine and five, when she is out of the office and when the schedule entries are marked "public". The subject information is that the rules apply to colleagues, the object information is that the entries must be marked "public" and the environment information is the time constraints and the user not being in the office. This generalized role-based access control allows context to be guarded by the situation in which it is being requested. In 6.3, we discuss the concept of situations in more detail.

Another cornerstone of this investigation is the ability to assign ownership to a piece of context or to a context event. When a user walks into a room and is identified, the user should own that information. But, if multiple users are in a meeting, the question of ownership is not so clear. The infrastructure could try to associate ownership to individual pieces of context or it could try and share ownership among the group. The former is quite difficult to perform manually and, thus, would be quite difficult to do automatically. The latter would require some changes to the system described above. For example, let us assume that a user not at the meeting has requested information about what was presented at a research group meeting. The user would provide an identity and the infrastructure would use that information to determine what subset of the requested information the user is entitled to. With shared ownership, the concerns of each of the owners must be taken into account. One solution might be to examine the relevant rules of each of the owners and apply the most restrictive one (*e.g.* one user may say that anyone can have access to this information, another may say that only members of the research group may have access and another may say that only her supervisor can have access – the last rule is the most restrictive and would be applied). Support for such a mechanism would have to be added.

An additional issue of this work deals with trust. At some level, a user of the context architecture must trust that the architecture is not deceptive, in much the same way as consumers trust credit card companies and the retailers at which they use their credit cards. Even with support for the mechanisms described in this section, there is still the potential that the architecture could behave incorrectly or malevolently. For example, a bug in the architecture implementation may allow restricted information to be delivered to unauthorized users. Or, the architecture could be designed to allow certain people access to whatever

information they want, regardless of rules specified by individual users. No matter what technological solution is applied in addressing the privacy and security concerns of using context, users must maintain a certain level of trust in the system or they will not use it. How we get our users to trust the context architecture enough to allow themselves to be both sensed and to share that information is an interesting and open question.

One method for addressing this problem is to show users what context is being sensed about them and their environment. As a user enters an instrumented environment, a privacy mirror (Everyday Computing Lab 2000) could display a reflection of what information is available about the user, much like a mirror reflects what can be visually seen about a user. The privacy mirror could also allow users to select which information should be public and which information should only be available to themselves. This is similar to the approach taken with Privacy Lamps and Vampire Mirrors (Butz, Beshers *et al.* 1998), where users in a virtual world could control what personal objects others could view.

6.2 Dealing with Inaccurate Context Data

In this section, we will discuss extensions to the Context Toolkit to deal with inaccurate context data being delivered from the context-sensing infrastructure to applications.

6.2.1 Motivation

Current context-aware services make the assumption that the context they are dealing with is correct. Our experience shows that though sensing is becoming more cost-effective and ubiquitous, it is still imperfect and will likely remain so. A challenge facing the development of realistic context-aware services, therefore, is the ability to handle imperfect context. There are ongoing attempts to improve context sensing to remove this imperfection or inaccuracy, including the design of more accurate sensors and the use of sensor fusion to combine the results of multiple homogenous or heterogeneous sensor readings (Brooks and Iyengar 1997). However, we believe that in most cases, no matter how much effort is placed behind these efforts, there are things that we will never be able to sense with 100% accuracy in a wide range of situations. And for those pieces of context that we can sense reliably and accurately, we still have the problem of making reliable and accurate inferences from them. What can we do in the face of inaccurate context data? The described Context Toolkit makes the (incorrect) assumption that the context it is receiving is 100% accurate, as do all of the infrastructures to support context-aware computing described in CHAPTER 2. Context-aware services that are built on top of these architectures act on the provided context without any knowledge that the context is potentially uncertain.

One approach that can be taken is to alert users that the context that is sensed about them, the context inferred about them or the context that they are using in an application is inaccurate. This would give them the opportunity to mediate or correct the context data before any irreversible action occurs. There are two existing systems that have taken this approach, the Remembrance Agent (Rhodes 1997; Rhodes 2000) and Multimodal Maps (Cheyer and Julia 1995).

The Wearable Remembrance Agent is a service available on a user's wearable computer that examines the user's location, identity of nearby individuals, and the current time and date to retrieve relevant information. The interpretation of the sensed context into relevant information is uncertain here. Rather than displaying the information with the highest calculated relevance, the Remembrance Agent instead presents the user with a list of the most relevant pieces of information and the relevance factor for each, on the user's head mounted display. In this way, the user can choose what is most relevant to the current situation from a filtered set.

The second service we will discuss is Multimodal Maps, a map-based application for travel planning. Users can determine the distances between locations, find the location of various sites and retrieve information on interesting sites using a combination of direct manipulation, pen-based gestures, handwriting and speech input. When a user provides multimodal input to the application, the application

uses multimodal fusion to increase the likelihood of recognizing the user's input. Rather than take action on the most likely input, if there is any uncertainty or ambiguity remaining after fusion, the application prompts the user for more information. By prompting the user for additional information, the system reduces the chance of making a mistake and performing an incorrect action.

Rather than assuming that sensed input (and its interpretation) is perfect, the two services demonstrate two techniques for allowing the user to correct uncertainty or *ambiguity* in implicitly sensed input. Note that both systems require explicit input on the part of the user before they can take any action. If a mobile user in a typical context-aware environment were forced to correct uncertainty in every piece of data that was sensed about her, she would be unwilling to use the system for very long as it would constantly be interrupting her asking for corrections to be made. The goal in our work is to provide an architecture that supports a variety of techniques, ranging from implicit to explicit, that can be applied to context-aware services. By removing the simplifying and incorrect assumption that all context is unambiguous, we are attempting to facilitate the building of more realistic context-aware services.

6.2.2 Mediation of Imperfect Context

Support for allowing users to correct (explicitly and implicitly) inaccurate or ambiguous context data is provided through the combination of the Context Toolkit and OOPS (Organized Option Pruning System) (Mankoff, Abowd *et al.* 2000; Mankoff, Hudson *et al.* 2000b; Mankoff, Hudson *et al.* 2000a), an architecture for the mediation of errors in recognition-based interfaces. But the solution is much more than the simple addition of the two systems. Imperfectly sensed context produces errors similar to recognition-based interfaces, but there are additional challenges that arise from the inherent mobility of humans in aware environments. Specifically, since users are likely to be mobile in an aware environment, the interactions necessary to alert them to possible context errors (from sensing or the interpretation of sensed information) and allow for the smooth correction of those errors must occur over some time frame and over some physical space.

First, we will discuss OOPS and describe how it is used to allow users to correct ambiguities that exist in desktop computing. Then, we will describe how the Context Toolkit and OOPS have been combined to allow for the disambiguation of data in context-aware situations.

6.2.2.1 OOPS

OOPS is a GUI toolkit that provides support for building interfaces that make use of recognizers (Mankoff, Abowd *et al.* 2000; Mankoff, Hudson *et al.* 2000b; Mankoff, Hudson *et al.* 2000a). Like implicit sensing, recognition is ambiguous, and OOPS provides support for tracking and resolving, or *mediating*, uncertainty in recognized input like speech and pen input. OOPS creates an internal model of recognized input, based on the concept of hierarchical events (Myers and Kosbie 1996), that allows separation of mediation from recognition and from the application. As we will see, this is a key abstraction that we use in the extended Context Toolkit. The OOPS model uses a directed graph to keep track of source input events, and their interpretations (which are produced by one or more recognizers). For example, when a user speaks, a speech recognizer may take the audio (the source event) and produce sentences as interpretations (child events). These sentences may be further interpreted, for example by a natural language system, as nouns, verbs, etc. Figure 42 shows the resulting graph.

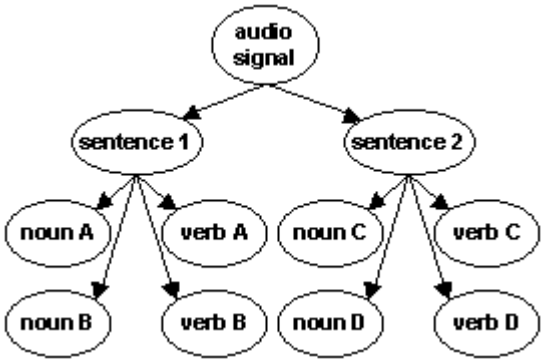


Figure 42: Hierarchical graph representing interpretations.

Note that there are two different sentences shown in this graph, at most one of which is correct (*i.e.* is what the user actually said). This situation, where there are multiple conflicting interpretations, is *ambiguous* and mediation is used to resolve the ambiguity. Rather than passing the ambiguously recognized input directly to an application (that is only interested in unambiguous input), the ambiguous input is passed to a mediator (that is only interested in ambiguous input). A mediator will display feedback about one or more interpretations to the user, who will then select one or repeat her input. Once the ambiguity is removed and the system's interpretation matches the user's intended input, OOPS updates the directed hierarchical graph to include information about which events were accepted (correct) and rejected (incorrect), and notifies the recognizer that produced the events and any consumers of the events (including applications), of what happened. At this point, consumers can act on the unambiguous events (for example, by executing the command specified by the user in his speech). Recognizers can use this information to update and improve their recognition ability.

To summarize, when recognizers return multiple interpretations of user input, OOPS automatically identifies ambiguity in this input and intervenes between the recognizer and the application by passing the directed graph to a mediator. The mediator initiates a dialog between the user and the system to remove any ambiguity in the input. Once the ambiguity is resolved, OOPS allows processing of the input to continue as normal.

6.2.2.2 Extending the Context Toolkit with OOPS

As stated previously, the Context Toolkit consists of widgets that implicitly sense context and deliver context to subscribers, interpreters that convert between context types, context handlers, applications that use context and a communications infrastructure that delivers context to these distributed components. OOPS consists of applications (interfaces) that produce input and deliver this input to subscribers, recognizers that convert between input types, input handlers and applications that use input. It is these parallels between the abstractions in the two systems that made it relatively easy to extend the Context Toolkit with the ambiguity and mediation concepts in OOPS. Few modifications to the Context Toolkit were required to allow us to support context-aware services that can deal with ambiguous context.

In order to understand our extensions, consider a single interaction. Initially, context is implicitly sensed by a context widget. Either this sensed context or interpretations of the sensed context can be ambiguous. A context interpreter is equivalent to a recognizer in OOPS and can create multiple ambiguous interpretations of sensed context. Each interpretation is an event that contains a set of attribute name-value pairs (a piece of context) and information about what it is an interpretation of (its source) and who produced it. The result of the interpretation is a directed graph, just like the representation used in OOPS (Figure 42). To deal with ambiguity, widgets were extended to keep track of the event graph containing the original piece of context and its interpretations, rather than just the original sensed context. Instead of passing only the context, which the widget's sensor or an interpreter provides, to subscribers, the widget passes the context

and its interpretations (the event graph) to its subscribers. It does not send the entire graph to subscribers, but also filters it to just those events that match the subscription request.

Since widgets and aggregators, as well as applications, may subscribe to a widget, all of these components must now include support for dealing with ambiguous events (and mediation of those events). A subscriber in the Context Toolkit receives data through an input handler (in BaseObject), which is responsible for dealing with distributed communications. The handlers in BaseObject were modified to check if the incoming context is ambiguous, and, if necessary, to send the context to a mediator instead of acting on the context immediately. BaseObject was modified to support the concept of mediators. Mediators are separate objects that are responsible for resolving the ambiguity in input, usually by creating a dialog with the user. They intervene between widgets (and aggregators) and applications in the same way that they intervene between recognizers and applications in OOPS.

The mediation of ambiguous context in the Context Toolkit is identical to the mediation of ambiguous user input in OOPS. Once a mediator provides feedback to the user, the user responds with additional input. The mediator uses this information to update the event graph. It does this by telling the widget that produced the events to accept or reject them as correct or incorrect. The widget then propagates the changes to its subscribers. If ambiguity is resolved, the events are delivered as normal and subscribers can act on them. Otherwise, the mediation process continues. Alternatively, a component may opt to receive events even when while they are ambiguous if it does not wish to wait for mediation to complete.

Because the Context Toolkit is a distributed system and because mediation is an interactive process that requires appropriate response times, only those portions of the event graph that have been subscribed to are passed across the network. This is done in order to minimize network calls (which can be quite extensive and costly) as a graph is generated and mediated. No single component can contain the entire graph being used to represent ambiguity. Providing each widget, aggregator and application with access to the entire graph for each piece of context and having to update each one whenever a change occurred (new event is added or an event is accepted or rejected) impedes the system's ability to deliver context in a timely fashion, as is required to provide feedback and action on context. However, if necessary, a component may request information about additional events.

To summarize the extensions to the Context Toolkit, all components were modified to handle the ambiguous event graph and not just single pieces of context information, handlers in BaseObject were modified to check for ambiguity and pass ambiguous context to mediators and BaseObject was extended to use the concept of mediators. Again, as foreshadowed in 4.2.4.1, context widgets made the support for dealing with ambiguous context easy to conceptualize and support. In the toolkit, context (and ambiguity) originates from widgets that send their information to subscribers so it is appropriate to start handling ambiguous information in the widgets, allowing them to send ambiguous information to their subscribers. Applications are not required to make significant changes to use this extended Context Toolkit, having only to state with each request for context whether or not they can deal with ambiguous information or not. If not, they can provide a set of mediators that can be used to resolve any ambiguity in the sensed and interpreted context.

6.2.3 Application: Mediating Simple Identity and Intention in the Aware Home

The application that we will describe is an In-Out Board installed in a home. The purpose of this service is to allow occupants of the home and others (who are authorized) outside the home to know who is currently in the home and when each occupant was last in the home, similar to the previously described In/Out Board application. This service may be used by numerous other applications as well. It is a piece of the Georgia Tech Broadband Residential Laboratory, a house that is being instrumented to be a context-aware environment (Kidd, Orr *et al.* 1999; Future Computing Environments 2000).

6.2.3.1 Physical Setup

Occupants of the home are detected when they arrive and leave through the front door, and their state on the In-Out Board is updated accordingly.

Figure 43 shows the front door area of our instrumented home, taken from the living room. In the photographs, we can see a small anteroom with a front door and a coat rack. The anteroom opens up into the living room, where there is a key rack and a small table for holding mail – all typical artifacts near a front door. To this, we have added two ceiling-mounted motion detectors (one inside the house and one outside), a display, a microphone, speakers, a keyboard and a dock beside the key rack.

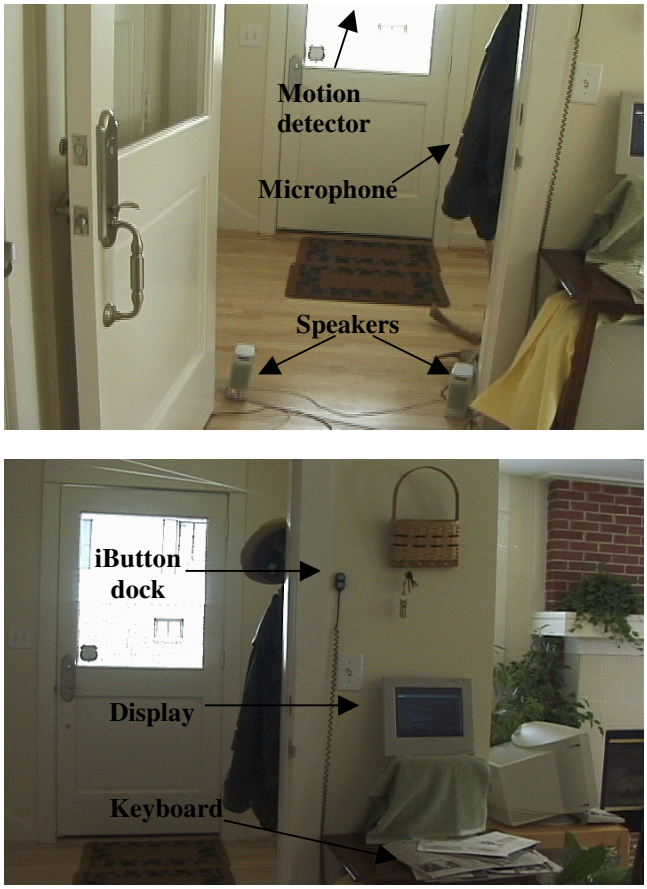


Figure 43: Photographs of In-Out Board physical setup.

When an individual enters the home, the motion detectors detect his presence. The current time, the order in which the motion detectors were set off and historical information about people entering and leaving the home is used to infer who the likely individual is and whether he is entering or leaving. This inference is indicated to the person through a synthesized voice that says “Hello Jen Mankoff” or “Goodbye Anind Dey”, for example. In addition, the wall display shows a transparent graphical overlay (see Figure 44) that indicates the current state and how the user can correct it if it is wrong: speak, dock or type. If the inference is correct, the individual can simply continue on as usual and the In-Out Board display will be updated with this new information.

Figure 44: In-Out Board with transparent graphical feedback.

If the inference is incorrect, the individual has a number of ways to correct the error. First, let us point out that the inference can be incorrect in different ways. The direction, identity or both may be wrong. The individual can correct the inference using a combination of speech input, docking with an iButton, and keyboard input on the display. These three input techniques plus the motion detectors range from being completely implicit to extremely explicit. Each of these techniques can be used either alone, or in concert with one of the other techniques. After each refinement, additional feedback is given indicating how the new information is assimilated. There is no pre-defined order to their use. Changes can continue to be made indefinitely, however, if the user makes no change for a pre-defined amount of time, mediation is considered to be complete and the service updates the wall display with the corrected input. The timeout for this interaction is set to 20 seconds.

For example, the user can say “No”, “No, I’m leaving/arriving”, “No, it’s Anind Dey”, or “No, it’s Anind Dey and I’m arriving/leaving”. The speech recognition is not assumed to be 100% accurate, so the system again indicates its updated understanding of the current situation via synthesized speech. Alternatively, an occupant can dock her iButton. The system then makes an informed guess based on historical information as to whether the user is coming or going. The user can further refine this using any of the techniques described if it is wrong. Finally, the occupant can use the keyboard to correct the input. By typing his name and a new state, the system’s understanding of the current situation is updated.

6.2.3.2 Application Architecture

We will now discuss how the architecture facilitated this service. The following figure (Figure 45) shows the block diagram of the components in this system.

Input is captured via context widgets that detect presence, using either the motion detectors, speech recognition, iButton or keyboard as the input-sensing mechanism. All of these widgets existed in the original Context Toolkit, but were modified to be able to generate ambiguous as well as unambiguous context information.

The motion detector-based widget uses an interpreter to interpret motion information into user identity and direction. The interpreter uses historical information collected about occupants of the house, in particular, the times at which each occupant has entered and left the house on each day of the week. This information is combined with the time when the motion detectors were fired and the order in which they were fired. A nearest-neighbor algorithm is then used to infer identity and direction of the occupant. The speech recognition-based widget uses a pre-defined grammar to determine identity and direction.

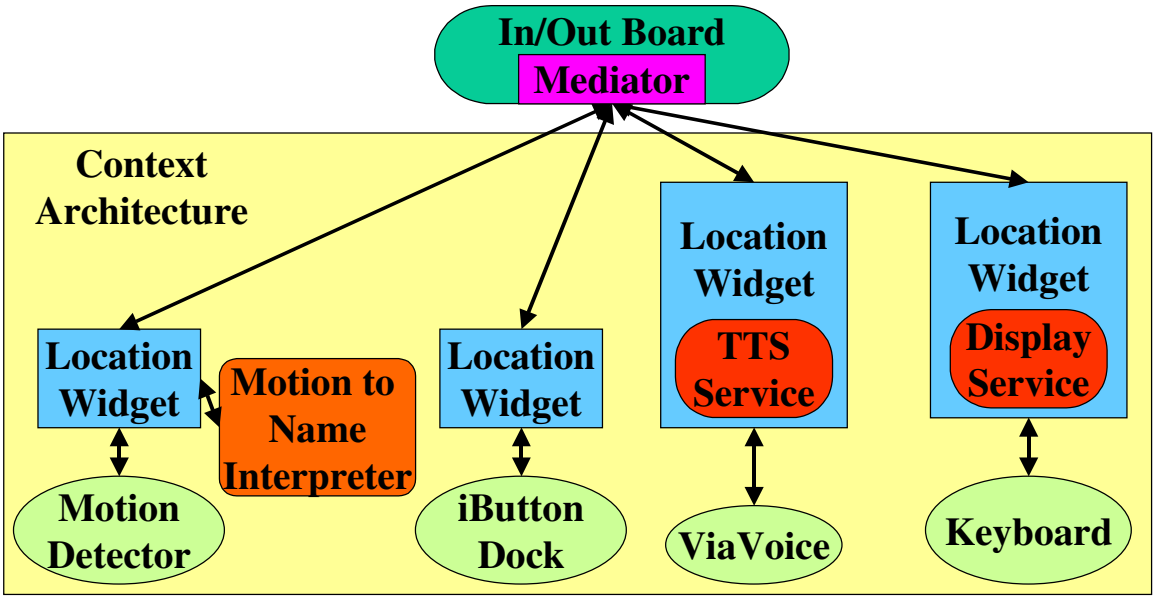


Figure 45: Architecture diagram for the In/Out Board application that uses a mediator to resolve ambiguous context.

When any of these widgets capture input, they produce not only their best guess as to the current situation, but also likely alternatives as well, creating an ambiguous event graph. The wall display has subscribed to unambiguous context information and is not interested in ambiguous information. When ambiguous information arrives, it is intercepted by a mediator that resolves it so that it can be sent to the application in its unambiguous form. The mediator uses this ambiguous information to mediate (accept and reject) or refine alternatives in the graph. The entire ambiguous graph is not held by any one component. Instead, it is distributed among the four context widgets and the mediator. Each component can obtain access to the entire graph, but it is not necessary in this service.

The mediator creates a timer to create temporal boundaries on this interaction. The timer is reset if additional input is sensed before it runs out. As the mediator collects input from the user and updates the graph to reflect the most likely alternative, it provides feedback to the user. It does this in two ways. The first method is to use a generic output service provided by the Context Toolkit. This service uses IBM ViaVoice to produce synthesized speech to provide feedback to the user. The second method is application-specific and is the transparent graphical overlay on the wall display shown in Figure 4. The transparent overlay indicates what the most likely interpretation of the user's status is and what the user can do to change their status: e.g. "Hello Anind Dey. Please dock, type, or speak if this is wrong." As the timer counts down, the overlay becomes more transparent and fades away.

When all the ambiguity has been resolved in the event graph and the timer has expired, the overlay will be faded completely and the correct unambiguous input is delivered to the wall display and the display updates itself with the new status of the occupant. Also, the input is delivered back to the interpreter so it has access to the updated historical information to improve its ability to infer identity and direction.

6.2.3.3 Design Issues

In this section, we will investigate the design heuristics that arose during the development of this service. The first issue is how to supply redundant mediation techniques. On the input side, in an attempt to provide a smooth transition from implicit to explicit input techniques, we chose motion detectors, speech, docking and typing. In order to enter or leave the house, users must pass through the doorway, so motion detectors are an obvious choice to detect this activity. Often users will have their hands full, so speech recognition is added as a form of more explicit, hands-free input. iButton docking provides an explicit input mechanism

that is useful if the environment is noisy. Finally, keyboard input is provided as an additional explicit mechanism and to support the on-the-fly addition of new occupants and visitors.

A valid question to ask is why not use sensors that are more accurate. Unfortunately in practice, due to both social and technological issues, there are no sensors that are both reliable and appropriate. As long as there is a chance that the sensors may make a mistake, we need to provide the home occupants with techniques for correcting these mistakes. None of the sensors we chose are foolproof either, but the combination of all the sensors and the ability to correct errors before applications take action is a satisfactory alternative.

On the output side, synthesized speech is used both to mirror the speech recognition input and to provide an output mechanism that is accessible (i.e. audible) to the user throughout the entire interaction space. Visual output for feedback is provided in the case of a noisy environment and for action as a persistent record of the occupancy state.

The next design decision is how to deal with the added complexity of distribution, over space and time, that arises in sensing input and supporting mediation in context-aware applications. The main question is where to place the input sensors and the rendering of the output to address the spatio-temporal characteristics of the physical space being used. There are "natural" interaction places in this space, where the user is likely to pause: the door, the coat rack, the key rack and the mail table. The input sensors were placed in these locations: motion sensors on the door, microphone in the coat rack, iButton dock beside the key rack and keyboard in a drawer in the mail table. The microphone being used is not high quality and requires the user to be quite close to the microphone when speaking. Therefore the microphone is placed in the coat rack where the user is likely to be leaning into when hanging up their coat. A user's iButton is carried on the user's key chain, so the dock is placed next to the key rack. The speakers for output are placed between the two interaction areas to allow it to be heard throughout the interaction space. The display is placed above the mail table so it will be visible to individuals in the living room and provide visual feedback to occupants using the iButton dock and keyboard.

Another design issue is what defaults to provide to minimize required user effort. We use initial feedback to indicate to the user that there is ambiguity in the interpreted input. Then, we leave it up to the user to decide whether to mediate or not. The default is set to the most likely interpretation, as returned by the interpreter. Through the use of the timeout, the user is not forced to confirm correct input and can carry out their normal activities. This is to support the idea that the least effort should be expended for the most likely action. The length of the timeout, 20 seconds, was chosen to allow enough time for a user to move through the interaction space, while being short enough to minimize between-user interactions.

We added the ability to deal with ambiguous context, in an attempt to make these types of applications more realistic. Part of addressing this realism is dealing with situations that may not occur in a prototype research environment, but do occur in the real world. An example of this situation is the existence of visitors, or people not known to the service. To deal with visitors, we assume that they are friendly to the system, a safe assumption in the home setting. That means they are willing to perform minimal activity to help keep the system in a valid state. When a visitor enters the home, the service provides feedback (obviously incorrect) about who it thinks this person is. The visitor can either just say "No" to remove all possible alternatives from the ambiguity graph and cause no change to the display, or can type in their name and state using the keyboard and add themselves to the display.

6.3 Extending the Context Toolkit: The Situation Abstraction

In the previous chapters, we described the Context Toolkit and how it helps application designers to build context-aware applications. We described the context component abstraction and showed how it simplified thinking about and designing applications. However, we also showed that the context component abstraction has some flaws that made the specification step of designing applications harder than it needs to be. The extra steps are:

- locating context widgets, aggregators and interpreters that are relevant to the application being built;
- deciding what combination of queries and subscriptions are necessary to acquire the context the application needs; and,
- collecting all the acquired context information together and analyzing it to determine when a situation, interesting to the application, has occurred.

A new abstraction called the *situation abstraction*, similar to the concept of a blackboard, makes these steps unnecessary. Instead, of dealing with components in the infrastructure individually, the situation abstraction allows designers to deal with the infrastructure as a single entity, representing all that is or can be sensed. Similar to the context component abstraction, designers need to specify what context their applications are interested in. However, rather than specifying this on a component-by-component basis and leaving it up to them to determine when the context requirements have been satisfied, the situation abstraction allows them to specify their requirements at one time to the infrastructure and leaves it up to the infrastructure to notify them when the request has been satisfied. This removes the three unnecessary steps listed above. By supporting the situation abstraction, designers receive the benefits of a blackboard model while building applications and the benefits of the component model while building components, as discussed in 4.2.4.1.

We will now use a hypothetical application to illustrate the differences in application design when using the context component abstraction and the situation abstraction. Following this, we will describe how the situation abstraction is implemented and supported by the Context Toolkit. We will then present CybreMinder, a prototype application that was built using the situation abstraction.

6.3.1 Motivation: Differences Between the Situation Abstraction and the Context Component Abstraction

The application that we will use to discuss the two abstractions is an extension to the Intercom application that we discussed in 5.3. In this case, a user doesn't want to speak with someone in her home, but wants to speak with her adult child Kevin, his wife and children who live in a separate home. The family is currently occupied so the user asks that the connection be made when the family is available. We will refer to this extended Intercom application as the Communications Assistant. It is a proposed application that facilitates the initiation of communications between remote locations. It uses sensed context and some intelligence (which we will treat as a black box for the purpose of this discussion) to automatically determine whether the connection between the caller and recipient should be made, based on the recipient's availability.

The decision on when to make the connection is complex, based on the importance of the communication, the recipient's location, identities of devices and people nearby the recipient, whether the recipient has free time (according to his schedule), the recipient's current activity and the history of the recipient. Using our example from above, one of the valid situations in which the connection can be made is when no one in the family is asleep or eating, when the adults have at least an hour of free time, when one adult and the children are home, and the two phone lines in the house and the cell phones are not being used.

When these conditions are met, the Communication Assistant uses information about where the adults are in the house, their recent location history and what they are doing to determine which phone to call. The conditions use simple context (location of individuals), context history (location information), context that needs interpretation (activities and which individuals are adults) and unrelated context, or context about different entities, (adults' schedules, family members' locations and activities and phone status) that must be combined in order to determine whether the call should be placed or not.

6.3.1.1 Building the Communications Assistant with the Context Component Abstraction

For this complex application, there are different combinations of queries and subscriptions that can be used to acquire the needed information. With this context component abstraction, the Context Toolkit does not naturally direct a programmer to use one particular combination. It is up to the programmer to specify the

logic required to handle context. Regardless of the combination used, the application must collect the requested context together to determine whether the desired circumstance has been realized.
For example, the application could locate and subscribe to all the relevant widgets:

- to the location widget with the condition that the information is about the family members and use an interpreter to determine whether the person is an adult or child
- to all of the phone widgets (assuming one for each phone) with the condition that the status of each phone is free
- to the activity widgets with the condition that the activity is not sleeping or eating
- to the schedule widget with the condition that no one in the family has an appointment in the next hour and use an interpreter to determine whether the appointment is for an adult.

When the application determines from the incoming context that the one of the adults and the children are in the house, that all the phones are free, that no one is eating or sleeping, that dinner has already been eaten, and that no adults are leaving the house in the next hour, it can query the location (for current and historical context) and activity widgets to determine which phone to call.

The same goal could be achieved by using a different set of queries and subscriptions. The application could locate all relevant widgets and then subscribe to the location widget to be notified when Kevin or his wife enters the house. When the application is notified of this event, it could subscribe again to the location widget to be notified when either of the children enters the house (and unsubscribe if Kevin or his wife leaves the house in the meantime). When the necessary occupants (one adult and both children) are in the house, the application can determine what phones are available within the house and subscribe to their representative phone widgets. If no phones are busy, the application can subscribe to the activity widgets to be notified if anyone is sleeping or eating. If a phone is used, the application could unsubscribe from the activity widgets. As well, when the necessary occupants are in the house, the application would subscribe itself to the family's schedule widget. As in the previous case, when the application determines that the correct set of circumstances has been sensed, it queries for the adults' location (current and historical) and activity information and then notifies the caller that she is being connected to her family.

The first set of queries and subscriptions is logically simpler for the application developer to provide (Figure 46).

1) Discovery: Use the resource discovery component to find a location widget that will notify us of people's location in the home, to find all the phone widgets in the house and to find the user's schedule widget.
2) Context Setup:
   a) Subscribe to the location widget with condition that the information is about the family members
   b) For each of the phone widgets, subscribe to be notified about the phone status
   c) Subscribe to each family member's activity to be notified about whether anyone is sleeping or eating
   d) Subscribe to the schedule widget to be notified with the condition that neither of the parents have a meeting in the next hour
   e) For each widget and condition (a-e), query for the current value and store
3) Context Handling:
   a) Check to see if the described circumstance is true using stored values. If so, go to 4.
   b) For each piece of context that comes in, request any necessary interpretation (if incoming information is about entrance or exit from the house, use interpreter to determine if person is spouse or child) and update stored values; repeat until the circumstance is true
4) Perform Behavior
   a) Query the activity widgets for history (has dinner been eaten?) and the location widgets for the parents' locations
   b) Connect the user with her family using the appropriate phone

Figure 46: Pseudocode for the Logically Simpler Combination of Queries and Subscriptions

However, it requires that the application handle unnecessary context information. For example, the application would be receiving data from all of the widgets in the house, when it really only needs to receive data from the widgets when the family is actually in the house. This requires the developer to provide additional code (essentially filters) to determine when the incoming context data is or is not relevant.

The second set of queries and subscriptions is logically more complex and harder for the programmer to specify, subscribing/unsubscribing to/from context widgets based on already received context, but it allows the application to perform less filtering or context handling that the first set needed (Figure 47).

1) Use the resource discovery component to locate all necessary widgets, and subscribe to a location widget that will notify us of the parents' locations with the condition that either parent is entering or exiting the house
2) When notified of a parent's entrance in the house, subscribe to this widget again, for any information about either child entering of leaving the house
3) When notified of a parent and both children being in the house, do the following:
   a) Subscribe to the phone widgets to be notified when phones are busy and query their status
   b) Subscribe to the parents' schedule widgets to be notified with the condition that the parent(s) in the home does not have to leave the house for at least one hour and query for their current data
4) When notified that no phones are busy, subscribe to the occupants' activity widgets and store all the context data and see if the circumstance can be satisfied; if so, go to step 6
5) When notified of new context:
   a) If both parents leave the house, unsubscribe from all widgets except that in step 1
   b) If either child is not in the house, unsubscribe from all widgets except that in steps 1 and 2
   c) If any phone is busy or a parent in the home has to leave the home in the next hour, unsubscribe from the activity widgets in step 4
   d) Update stored values and check to see if the described circumstance is true
6) Perform Behavior
   a) Query the activity widget for its history and the location widget for the parents' locations
   b) Connect the user to her family on the appropriate phone

Figure 47: Pseudocode for the More Complex Combination of Queries and Subscriptions

To summarize, the required support for the application using the context components abstraction, follows:

- Acquisition of context from the sensor (if widgets are not available);
- Locating the necessary widgets, interpreters and aggregators; and,
- Providing the logic needed to handle context, including:
  - Requesting the desired context through a series of queries and subscriptions; and,
  - Collecting the requested context together to determine whether the correct circumstance has been sensed.

6.3.1.2 Building the Communications Assistant with the Situation Abstraction

Here we will show that if we use an abstraction that is built on top of the context component abstraction, we will, predictably, make it even easier for developers to build applications. This abstraction is called the *situation* abstraction. For the Communications Assistant, the developer specifies a situation that directs the context-sensing infrastructure to notify the application when no one in the family is asleep or eating, when the adults have at least an hour of free time, when one adult and the children are home, and the two phone

lines in the house and the cell phones are not being used. This is quite similar in spirit to what was done when using the context component abstraction. When the Communications Assistant is notified of such a situation, it queries the infrastructure (and not a particular component) for the parents' current location and location history and the parents' current activity and uses this information to determine which phone to use to complete the desired connection.

The difference is that when specifying a situation, the application communicates with the context-sensing infrastructure as a whole (as shown by the box separating the Communications Assistant from the context architecture in Figure 48b), rather than just communicating with a single context component at one time (as shown in Figure 48a). To be more specific, in the situation abstraction, the programmer declaratively specifies that the application wants to know about the discussed situation and the infrastructure notifies the application when this occurs. In the context component abstraction, the programmer had to determine what components can provide the necessary context, what combination of queries and subscriptions to use, subscribe to them directly, wait to be notified of context by each component and then combine the received context to determine when the desired situation has been achieved. The situation abstraction does not leave the details of locating the necessary components, defining the logic for handling context or for combining context to determine when situations are realized to the programmer and, instead, provides this automatically for the programmer.

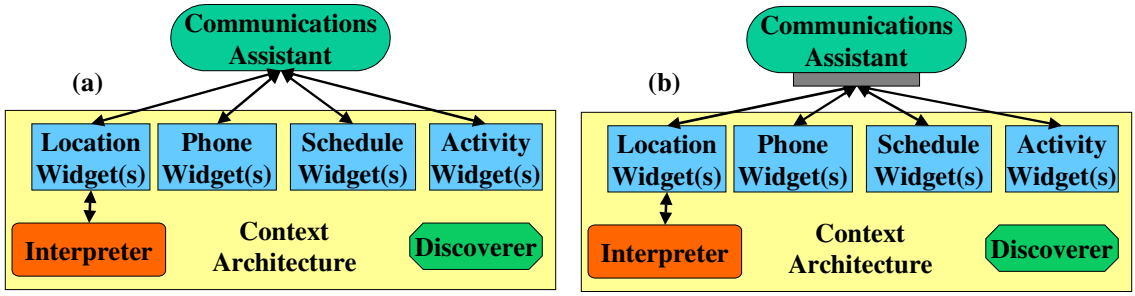


Figure 48: Architecture diagram for the Communications Assistant, using both the (a) context component abstraction and the (b) situation abstraction.

To summarize, the required support for the application using the situation abstraction, follows:

- Acquisition of context from the sensor (if widgets are not available); and,
- Specifying the situation and the context history that the application is interested in.

6.3.1.3 Additional Context-Aware Application Development Concerns

In this section, we will discuss how use of these abstractions differs in the presence of added components, failing components, evolving applications and multiple applications.

6.3.1.3.1 Adding Sensors

If new sensors are added to the existing system, existing applications should be able to take advantage of them if they provide context that the application is using. When there is no programming abstraction being used, programmers have to provide all the necessary support described before to allow the sensor to be used. When the component or situation abstraction is being used, programmers have the easier task of only specifying how context can be acquired from the new sensor and not having to worry about storage, distribution, and the other issues discussed earlier. Users of the component abstraction have to do a little work to use added sensors, but users of the situation abstraction can use them automatically with no extra effort required.

We will revisit the Communications Assistant application to make this more concrete. One of the widgets that the application uses is a Location Widget to indicate when people enter or leave the home. We will assume that the widget uses an intelligent floor tile (Orr 2000) to determine the identities of people entering

and leaving. If a new sensor wrapped in a context widget is added, say a face recognizer, then the application has a new widget that it could potentially use to provide more information. If no abstraction support (situation or component) were provided, then the application would not know that a new component was available. The designer would have to stop the application, modify it to take advantage of the new sensor and then restart it.

With the component abstraction, the application can take advantage of the new sensor, only if the designer gave it the ability to do so. The application must use the discovery mechanism to indicate its interest in the location context, subscribing to the Discoverer to find any new component that can provide relevant location information. When a new component becomes available that can provide some of this context, the application would be notified and would then use it accordingly (subscribe or query to a widget, interpret with an interpreter). The designer must explicitly provide this ability in the application.

With the situation abstraction, the application can take advantage of the new sensor automatically, without any intervention on the programmer's part. Because the situation abstraction deals with the entire context-sensing infrastructure as a whole and not with individual components, it is able to automatically support the handling of changes in the infrastructure. The discovery mechanism provides notification of the new sensor (widget) being added, allowing the application-specified situation to make use of it.

6.3.1.3.2 Failing Components

Ideally, when a component fails in the system, the application should automatically switch to using another appropriate component or, if no such component exists, be notified that the desired context can no longer be sensed. Again, using the Communications Assistant, we will take the example of the intelligent floor widget and have it fail. When there is no abstraction being used, it is entirely up to the programmer to provide the capabilities to detect that the widget failed, to find a new component that can provide the same location information and communicate with it.

With the component abstraction, an application will be notified when the widget it is using has failed. However, it is still up to the application programmer to provide the runtime capabilities to determine whether there is another widget that can provide the desired context, via the Discoverer, and to subscribe to it. If the situation abstraction is used, the ideal case is supported. The situation that an application wants to be notified of will automatically switch from using the failed widget to using another widget that provides equivalent context. No extra effort is required on the part of the designer. If no such component exists, the application is notified that its request can no longer be fulfilled.

6.3.1.3.3 Evolving Applications

Context-aware applications can be evolved to use new sensors, as discussed in section 6.3.1.3.1, or to use a different set of context information. We will now discuss this latter case. Suppose that we want to modify the Communications Assistant's complex situation to include a condition that the house temperature must be greater than 75 degrees, in order to place a call for the user. When there is no abstraction being used, it is completely up to the programmer to determine how to acquire the temperature context from the environment. This may also involve providing new sensors and new support for using those sensors. With the context component abstraction, the programmer must also determine the new set of queries and subscriptions (in this case, probably only a single subscription) that are required to obtain the desired temperature information. The greater the difference between the original set of context and the new set of context, the greater the effort required by the programmer. But, with the situation abstraction, the programmer need only to specify what the new context required is, and the existing infrastructure takes on the responsibility of providing this context to the application.

6.3.1.3.4 Multiple Applications

All the discussion in this chapter has focused on the existence of a single application. When multiple applications are added that need to use the same context as our original application, then there needs to be

support to allow the applications to share the context-sensing infrastructure. Let us add a second application, an intelligent security system that uses this infrastructure. The security system automatically arms itself when there is no one in the house and deactivates itself when a known adult is in the house. Using the component and situation abstractions, both applications automatically have access to the context-sensing infrastructure. When there is no programming abstraction, the coupling of the original application to the context-sensing infrastructure is very important. If there is tight coupling, then the programmer needs to do quite a bit of work to loosen the coupling to allow the delivery of context to the smart alarm system. If there is loose coupling, the programmer may have less work to do to separate the infrastructure from the application and support remote delivery of context to the smart alarm system.

6.3.1.4 Abstraction Summary

The following table summarizes the features that are supported by no programming abstraction, the component abstraction and the situation abstraction. It should be clear that the situation abstraction provides the most support to context-aware application programmers. Why would a programmer ever use the context component abstraction when the situation abstraction is available? There is a tradeoff in the three abstractions between the amount of underlying detail that is known by the application designer and the ease by which the designer can build applications (due to the support of common features). If a designer wants control over how the queries, interpretations and subscriptions are performed, then she probably wants to use the component abstraction. If these details are not important, than the situation abstraction is more appropriate. It should be noted that while most details are initially hidden from designers using the situation abstraction, designers can request any information about the underlying details, as needed.

Table 4: Summary of the feature support provided by each programming abstraction.

✕ = no support, P = partial support, ✔ = complete support

Features	Abstractions		
	None	Component	Situation
Context acquisition	✕	P	P
Distributed communications	✕	✔	✔
Querying/subscription	✕	✔	✔
Storage	✕	✔	✔
Context specification	✕	P	✔
Situation realization	✕	✕	✔
Sensor/component addition	✕	P	✔
Sensor/component failure	✕	P	✔
Support for evolving applications	✕	P	✔
Support for multiple applications (reuse)	✕	✔	✔

The component abstraction provides the benefits of the component representation of the world (discussed in 4.2.4.1) while the situation abstraction provides the benefits of the blackboard representation. By supporting both the component abstraction and the situation abstraction and allowing them to be interoperable, designers have the choice of which abstraction to use at any time and can even use them at the same time in an application, if desired.

6.3.2 Implementation of the Situation Abstraction

The main difference between using the context component abstraction and the situation abstraction is that in the former case, applications are forced to deal with each relevant component individually, whereas in the latter case, while applications can deal with individual components, they are also allowed to treat the context-sensing infrastructure as a single entity.

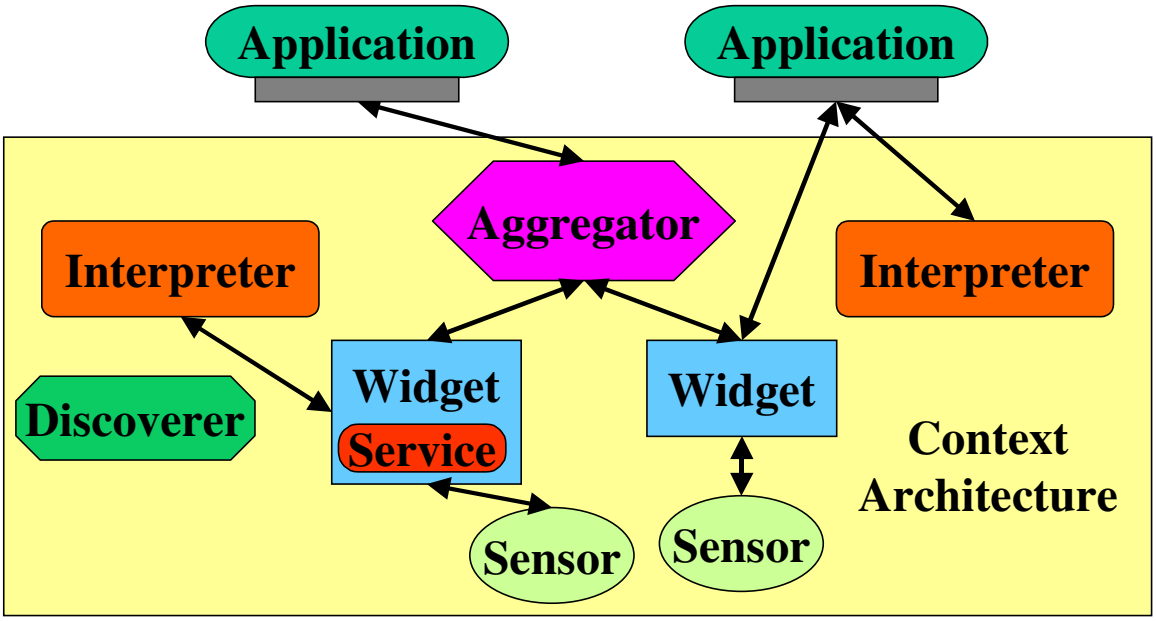


Figure 49: Typical interaction between applications and the Context Toolkit using the situation abstraction.

Figure 49 shows how an application can use the situation abstraction. It looks quite similar in spirit to Figure 8. The difference is that from the application's standpoint, it is not dealing with individual widgets, interpreters and aggregators, but with the infrastructure as a whole. With the situation abstraction, the application can specify a single situation or request for context that spans multiple components. Rather than the application designer having to determine what set of subscriptions and interpretations must occur for the desired context to be acquired, it hands this job off to an extended BaseObject (shown in Figure 49, sitting between the application and the context architecture). BaseObject determines what subscriptions and interpretations are required (with the help of a Discoverer) and interacts with the infrastructure to make it happen. First the application specifies what context situation it is interested in, using its BaseObject instance. The situation consists of a number of conditions, where each condition consists of a context type, an operator and a value, as defined in Figure 12. For example "Anind Dey is present in Room 383" equates to "username = Anind Dey AND location = Room 383" and "stock price of Coca-Cola is greater than $65" equates to "stockname = Coca-Cola AND price > 65".

BaseObject takes a situation and determines what set of subscriptions and interpretations must occur to realize it. It uses the algorithm in Figure 50 to determine the set.

Basically, a mapping between the specified situation conditions and the available widget/aggregator callbacks and interpreter interpretations (collected from the Discoverer) is made. Let us first assume that no interpretation is required by the specified situation. For each condition or sub-situation in the situation, the algorithm attempts to find at least one callback provided by a widget or aggregator that could potentially satisfy that condition. If it finds multiple callbacks, the default case is to choose one (although by setting a flag, multiple callbacks can be chosen).

We will now discuss how the algorithm chooses one out of multiple potential satisfying callbacks. If an application creates a situation that contains the condition "anyone is present in Room 383" (equates to username = * AND location = Room 383"), for example, then any callback that contains the attributes "username" and "location" could satisfy this request. However, any of these callbacks that contain the constant attribute "location = Room 383" would be a more appropriate choice because it more closely matches the requested information. The algorithm chooses a callback that has the most constant attributes

that match those in the condition. If there are multiple callbacks with an equal number of relevant constant attributes, the algorithm favors aggregators over widgets because interacting with aggregators requires fewer network connections. If there are still multiple candidates, the first candidate found is chosen.

1) Call Generate_Relationships
2) If no errors, query the widgets/aggregators and call interpreters to determine if the situation is currently true
3) If so, notify the application but if not, subscribe to widgets/aggregators returned
4) Set the state for each sub-situation to be unsatisfied and handle returned data (Handle below)

Generate_Relationships
For each sub-situation in the situation,
   a) Find component(s) that can fulfill clause (Find_Components below)
   b) If any, return them; else return an error

Find_Components
1) Find all widgets/aggregators that match clause attributes in sub-situation
2) If any found, return best match, choosing aggregators over widgets and components with the most constant attribute matches
3) If none found, make list of partial matches, ordered by the maximum number of matches
4) For each of these partial matches:
   a) Look for collections of interpreters whose combination of outputs are a superset of the unmatched attributes that takes matched attributes and returns unmatched attributes (disallowing any combinations that use cycles of interpreters)
   b) For each collection of interpreters, take the list of input attributes for the interpreters and call Generate_Relationships
   c) If none, return error; else choose the collection that uses the fewest number of components and keep track of the combination of widgets/aggregators and interpreters
5) Return the combination of widgets/aggregators and interpreters for the sub-situation

Handle
1) When new data arrives, hand off data to appropriate sub-situation handler
2) Sub-situation handler updates its data and calls any necessary interpreters (as specified by the combination returned from Find_Components)
3) With the widget/aggregator data and any interpretation data, determine if the sub-situation has been satisfied
4) If so, set the sub-situation state to satisfied and check if all the sub-situation states are satisfied
5) If all are satisfied, notify the application

Figure 50: Algorithm for converting situation into subscriptions and interpretations.

If interpretation can be used to satisfy the condition, the algorithm favors no interpretation to using interpretation. If no callback can be found that satisfies the condition, combinations of callbacks and interpretations are examined to find a combination that will satisfy the condition. Using our example, a combination of a callback that contains the attributes “username” and “buildingCoordinates” and an interpreter that converts between “buildingCoordinates” and “location” would satisfy the condition. The

algorithm will look for a chain of interpretations (and callbacks) if a single interpretation will not suffice (*e.g.* GPS coordinates to city coordinates to building coordinates to location).

If no mapping can be made between the specified situation and the available components in the context-sensing infrastructure, the application is notified. If the mapping can be made, BaseObject queries each of the components it is going to subscribe to, to determine if the situation is currently true. If it isn't true, BaseObject automatically subscribes to these components and the relevant callbacks, as specified by the mapping. When a callback notifies BaseObject of a change in value, BaseObject performs any necessary interpretations (to convert the received context into the form required by the situation) and then re-evaluates the situation to see whether it can be satisfied. When the situation is satisfied, the application is notified.

When a mapping has been made and subscriptions set up, components can still be added and components can fail. When either situation happens, the Discoverer notifies the BaseObject and the algorithm is executed again. This allows a new mapping to be created that can use the changed context-sensing infrastructure.

If the original mapping could not be created or a new mapping could not be created (in the face of component failures), the application is notified immediately. However, there is a case where the infrastructure believes that it can satisfy the situation, but actually cannot. Take the following example where a situation is created containing the condition "Anind Dey is in Room 383". There may be a callback that contains the attribute "username" and constant attribute "location = Room 383", but the widget/aggregator may not be able to sense Anind Dey's presence. The range of usernames that the component can return does not include "Anind Dey", but this information is not known outside the component. This information, called *coverage*, is part of a larger issue we refer to as *quality of service*, and this issue will be discussed as part of our future work.

6.3.3 CybreMinder: A Complex Example that Uses the Situation Abstraction

Whereas the Communications Assistant is an example of how a programmer can use the situation abstraction to build applications, the CybreMinder application, which we will describe in this section, is an example of how end-user can use the situation abstraction. The CybreMinder application is a prototype application that was built to help users create and manage their reminders more effectively (Dey and Abowd 2000a). Current reminding techniques such as post-it notes and electronic schedulers are limited to using only location or time as the trigger for delivering a reminder. In addition, these techniques are limited in their mechanisms for delivering reminders to users. CybreMinder allows users to specify more complex situations and associate these with reminders. The complex situations allow for a more appropriate trigger than those based solely on location and time. When these situations are realized, the associated reminder will be delivered to the specified recipients. The recipient's context is used to choose the appropriate mechanism to use for delivering the reminder.

6.3.3.1 Creating the Reminder and Situation

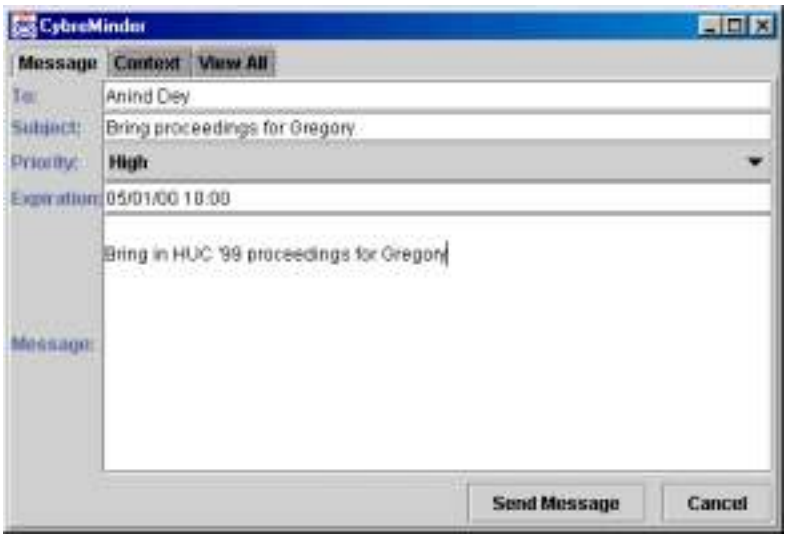


Figure 51: CybreMinder reminder creation tool.

When users launch CybreMinder, they are presented with an interface that looks quite similar to an e-mail creation tool. As shown in Figure 51, users can enter the names of the recipients for the reminder. The recipients could just be themselves, indicating a personal reminder, or a list of other people, indicating a third party reminder is being created. The reminder has a subject, a priority level (ranging from lowest to highest), a body in which the reminder description is placed, and an expiration date. The expiration date indicates the date and time at which the reminder should expire and be delivered, if it has not already been delivered.

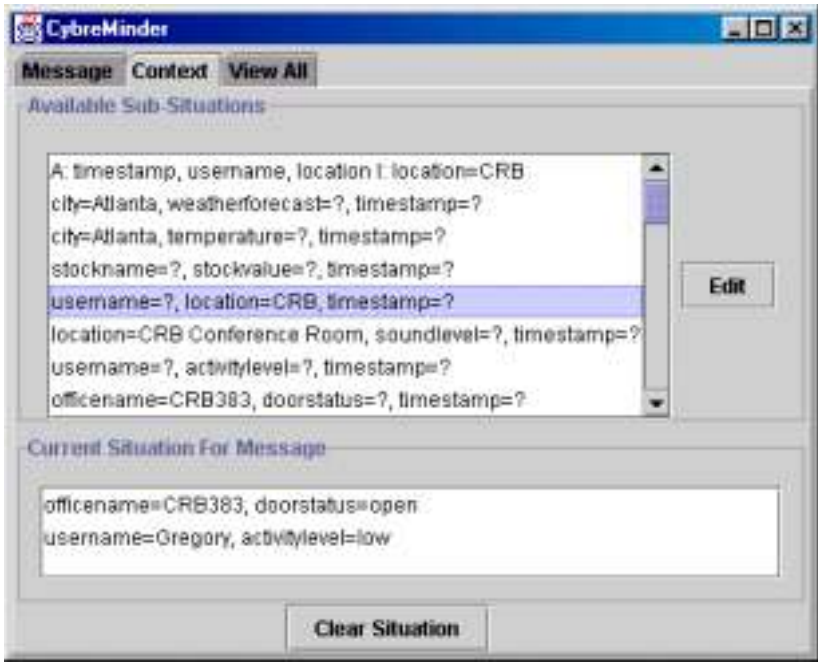


Figure 52: CybreMinder situation editor.

In addition to this traditional messaging interface, users can select the Situation tab and be presented with the situation editor (Figure 52). This interface allows dynamic construction of an arbitrarily rich situation, or context that is associated with the reminder being created. The interface consists of two main pieces for creating and viewing the situation. Creation is assisted by a dynamically generated list of valid sub-situations that are currently supported by the CybreMinder infrastructure (as assisted by the Context Toolkit described later). When the user selects a sub-situation, they can edit it to fit their particular situation. Each sub-situation consists of a number of context types and values. For example, in Figure 52, the user has just selected the sub-situation that a particular user is present in the CRB building at a particular time. The context types are the user's name, the location (set to CRB) and a timestamp.

In Figure 53, the user is editing those context types, requiring the user name to be "Anind Dey", and not using time. This sub-situation will be satisfied the next time that Anind Dey is in the location 'CRB'.

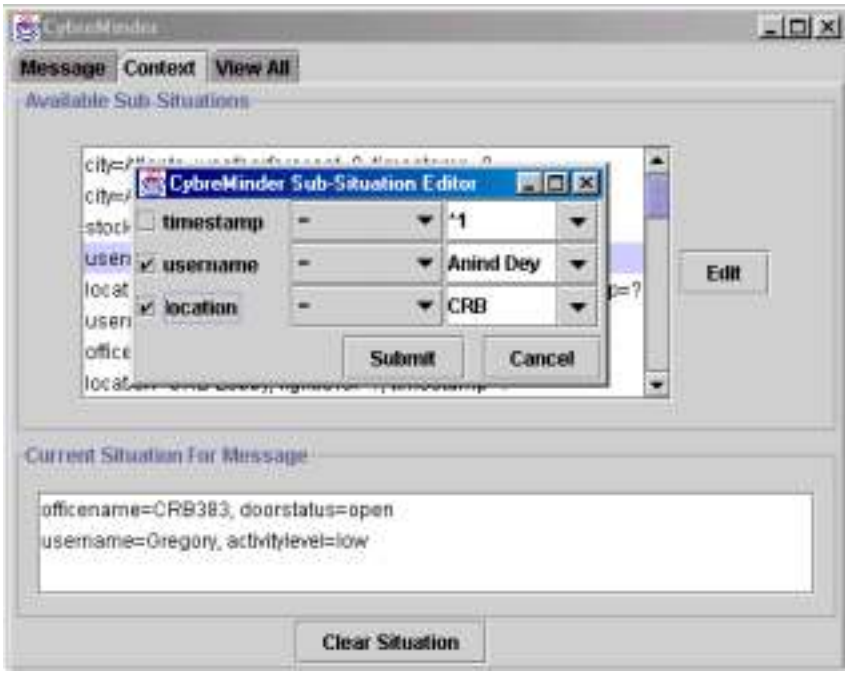


Figure 53: CybreMinder sub-situation editor.

The user indicates which context types are important by selecting the checkbox next to those attributes. For the types that they have selected, users may enter a relation other than '='. For example, the user can set the timestamp after 9 p.m. by using the '>' relation. Other supported relations are '>=', '<', and '<='. For the value of the context, users can either choose from a list of pre-generated values, or enter their own.

At the bottom of Figure 53, the currently specified situation is visible. The overall situation being defined is the conjunction of the sub-situations listed. Once a reminder and an associated situation have been created, the user can send the reminder. If there is no situation attached, the reminder is delivered immediately after the user sends the reminder. However, unlike e-mail messages, sending a reminder does not necessarily imply immediate delivery. If a situation is attached, the reminder is delivered to recipients at a future time when all the sub-situations can be simultaneously satisfied. If the situation cannot be satisfied before the reminder expires, the reminder is delivered both to the sender and recipients with a note indicating that the reminder has expired.

6.3.3.2 Delivering the Reminder

Thus far, we have concentrated on the process of creating context-aware reminders. We will now describe the delivery process. When a reminder can be delivered, either because its associated situation was satisfied or because it has expired, CybreMinder determines what is the most appropriate delivery mechanism for each reminder recipient. The default signal is to show the reminder on the closest available display, augmented with an audio cue. However, if a recipient wishes, they can specify a configuration file that will override this default.

A user's configuration file contains information about all of the available methods for contacting the user, as well as rules defined by the user on which method to use in which situation. If the recipient's current context and reminder information (sender identity and/or priority) matches any of the situations defined in his configuration file, the specified delivery mechanism is used. Currently, we support the delivery of reminders via SMS on a mobile phone, e-mail, displaying on a nearby networked display (wearable, handheld, or static CRT) and printing to a local printer (to emulate paper to-do lists).

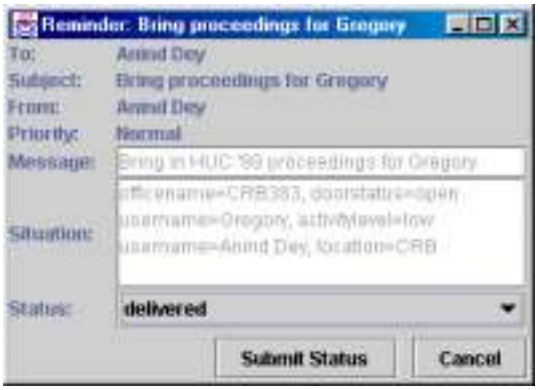


Figure 54: CybreMinder display of a triggered reminder.

For the latter three mechanisms, both the reminder and associated situation are delivered to the user. Delivery of the situation provides additional useful information to the user, helping them understand why the reminder is being sent at this particular time. Along with the reminder and situation, users are given the ability to change the status of the reminder (Figure 54). A status of "completed" indicates that the reminder has been addressed and can be dismissed. The "delivered" status means the reminder has been delivered but still needs to be addressed. A "pending" status means that the reminder should be delivered again when the associated situation is next satisfied. Users can explicitly set the status through a hyperlink in an e-mail reminder or through the interface shown in Figure 55.

Figure 55: List of all reminders.

Since SMS messages have limited length, only the subject of a reminder is delivered when using this delivery mechanism. Users receiving such an SMS message have the option of going to a networked device and launching their interface (Figure 51) to CybreMinder. By selecting the View All tab, users can view a personalized list of all reminders and can change the status of any of these reminders (Figure 55).

The CybreMinder application is the first application we built that used the situation abstraction. It supports the automatic execution of services, both in providing a list of what can be sensed in the environment and in delivering reminders to users.

6.3.3.3 Example Reminders

In this section, we will describe a range of reminders and situations that users can create using CybreMinder, starting from simple situations and moving towards more complex situations. The situations are only limited by the context that can be sensed. Table 5 gives the natural language and CybreMinder descriptions of the illustrated situations below.

Table 5: Natural language and CybreMinder descriptions of reminder scenarios.

Situation	Natural Language Description	CybreMinder Description
Time	9:45 am	Expiration field: 9:45 am
Location	Forecast is for rain and Bob is leaving his apartment	City = Atlanta, WeatherForecast = rain Username = Bob, Location = Bob's front door
Co-Location	Sally and colleague are co-located	Username = Sally, Location = *1 Username = Bob, Location = *1
Complex #1	Stock price of X is over $50, Bob is alone and has free time	StockName = X, StockPrice > 50 Username = Bob, Location = *1 Location = *1, OccupantSize = 1 Username = Bob, FreeTime > 30
Complex #2	Sally is in her office and has some free time, and her friend is not busy	Username = Sally, Location = Sally's office Username = Sally, FreeTime = 60 Username = Tom, ActivityLevel = low

6.3.3.3.1 Time-Based Reminder

Like many of the other systems previously described, CybreMinder allows reminders to be triggered based on a simple time context. In this scenario, Sally has a meeting at 10 a.m. tomorrow. She wants to send a reminder to herself fifteen minutes before the meeting occurs, so that she has time to walk to the meeting. She can simply set the expiry date to be tomorrow's date and 9:45 a.m.

6.3.3.3.2 Location-Based Reminder

In this scenario, Bob wants to remind himself to take his umbrella to work because it is supposed to rain this afternoon. He keeps the umbrella near his apartment door, so he wants to receive the reminder as he approaches the door. Here, he can simply create a situation with only one sub-situation: he is at his front door. In CybreMinder terms, he sets the username to his name, and location to be his front door. This situation can be made slightly more complex. If Bob is sending the reminder the night before, then he may want to add a time attribute and set it to be greater than 7:00 a.m. By doing so, the reminder will not be triggered and displayed each time he goes in and out of his apartment that night. It will only be displayed when he approaches the door after 7:00 a.m. the next morning.

Pushing on this scenario a little more, Bob does have to know ahead of time that it is going to rain. He can simply create a reminder that is to be delivered whenever the forecast calls for rain and he is leaving his apartment.

6.3.3.3.3 Co-location-Based Reminder

Of the reminder systems we reviewed, only Proem (Kortuem, Segall *et al.* 1999) supported proactive reminders when two or more people were co-located in an arbitrary location. It can be argued that post-it notes could be used in this setting, although it currently breaks normal social conventions to stick post-it notes to people. An example co-location scenario follows: Sally wants to engage a colleague in a discussion about an interesting paper she read recently, but she does not remember this when she sees her colleague. She can create a context-aware reminder that will be delivered when she is in close proximity with her colleague. The situation she creates is slightly more complex than the ones we have discussed so far, and it makes use of variables. Variables allow users to create relationships between sub-situations. First Sally creates an initial sub-situation where she sets the user name to be her colleague's name and the location to be variable (indicated in Table 1 by *1). Then, she creates a second sub-situation, where she sets the user name to be her name and the location to the variable used in the first sub-situation. Now when Sally and her colleague are in the same arbitrary location, the reminder will be delivered.

6.3.3.3.4 Complex Reminder

CybreMinder supports the unlimited use of rich context, allowing users to create as rich a situation as can be sensed. We will describe two such situations. In the first scenario, Bob owns stock in Company X and he has decided to sell that stock when the stock is valued over $50 per share. He only wants to be reminded to sell, however, when he is alone and has some free time. To create this situation to signal a reminder to sell, Bob creates a number of sub-situations: stock price of company X > $50, Bob is the only occupant of his location, and Bob's schedule shows that he has > 30 minutes before his next meeting. When this situation occurs, Bob receives the reminder to sell his stock.

In our second complex scenario, Sally needs to make a phone call to her friend Tom. She wants to receive a context-aware reminder when she arrives at her office, has some free time in her schedule, and her friend is not busy. To create this situation, she creates three sub-situations: Sally is in her office, Tom's activity status is low, and Sally has at least one hour before her next appointment.

6.3.3.4 Building the Application

The context architecture used to build CybreMinder is shown in Figure 56. For this application, the architecture contains a user aggregator for each user of CybreMinder and, because the application is context-independent, any combination of widgets, aggregators and interpreters. When CybreMinder launches, it makes use of the discovery protocol in the Context Toolkit to query for the context components are currently available to it. It analyzes this information and determines what sub-situations are available for a user to work with. The sub-situations are simply the collection of subscription callbacks that all the context widgets and context aggregators provide. For example, a Presence context widget contains information about the presence of individuals in a particular location (specified at instantiation time). The callback it provides contains three attributes: a user name, a location, and a timestamp. The location is a constant, set to "home", for example. The constants in each callback are used to populate the menus from which users can select values for attributes.

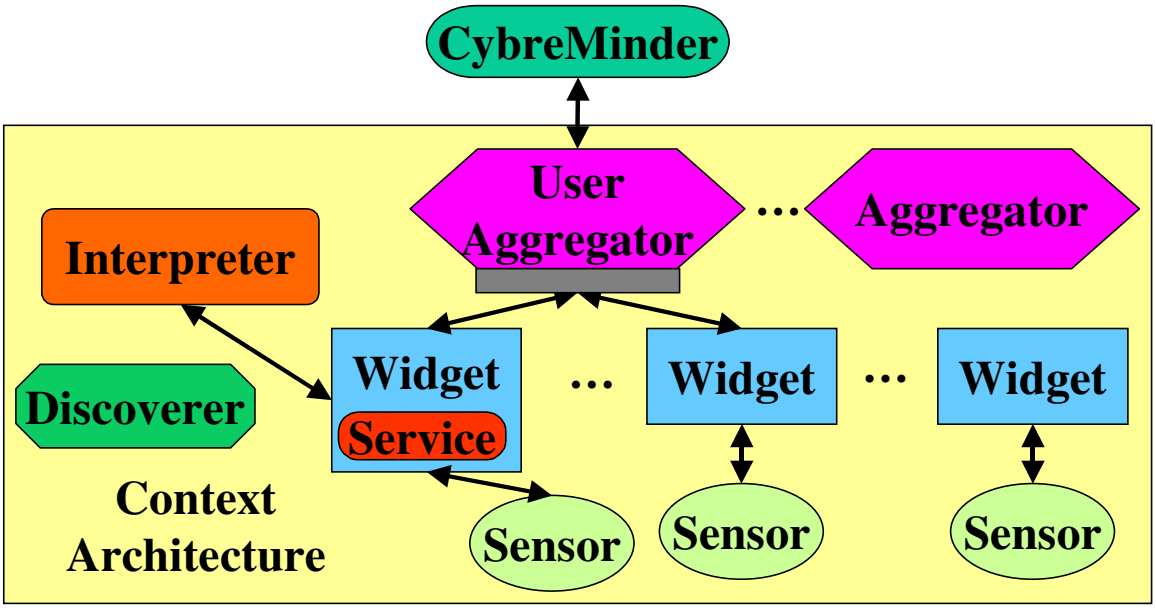


Figure 56: Architecture diagram for the CybreMinder application, with the User Aggregator using the extended BaseObject.

When the user creates a reminder with an associated situation, the reminder is sent to the aggregator responsible for maintaining context about the recipient – his User aggregator. CybreMinder can be shut down any time after the reminder has been sent to the recipient's aggregator. The recipient's aggregator is the logical place to store all reminder information intended for the recipient because it knows more about the recipient than any other component and is always available. This aggregator analyzes the given situation and creates subscriptions to the necessary aggregators and widgets (using the extended BaseObject) so that it can determine when the situation has occurred. In addition, it creates a timer thread that awakens when the reminder is set to expire. Whenever the aggregator receives a subscription callback, it updates the status of the situation in question. When all the sub-situations are satisfied, the entire situation is satisfied, and the reminder can be delivered.

The recipient's aggregator contains the most up-to-date information about the recipient. It tries to match this context information along with the reminder sender and priority level with the rules defined in the recipient's configuration file. The recipient's context and the rules consist of collections of simple attribute name-value pairs, making them easy to compare. When a delivery mechanism has been chosen, the aggregator calls a widget service that can deliver the reminder appropriately. For example, a display widget provides information about the display capabilities of a device. It also provides a service that allows other components to display information on that device. Similarly, e-mail and SMS services exist in the Context Toolkit.

Services can also return information to the component that calls them. For example, the display service not only shows the reminder and associated situation, but also a form allowing the user to set the state of this reminder (Figure 54). The user input to this form is sent back to the recipient's aggregator, which can update the reminder status. In the case of SMS, when the user must set the status using the CybreMinder, the application contacts the user's aggregator and queries for all the reminders and associated information. The application sends any updated status information to the aggregator.

6.3.3.5 Toolkit Support

The CybreMinder application makes use of the situation abstraction. It allows end users to specify the situations they are interested in, which the application then translates into a system specification of the situations. By leveraging off of the Context Toolkit's ability to acquire and distribute context, we allow users to create arbitrarily complex situations to attach to reminders and to create custom rules for governing how reminders should be delivered to them. Users are not required to use templates or hardcoded situations, but can use any context that can be sensed and is available from their environment. By using a discovery protocol for determining what context is available, we attempt to limit users in creating situations that CybreMinder is able to detect. If we generalize the end user's use of the situation abstraction, from delivering messages when a specified situation occurs to performing some arbitrary specified action, we can support powerful end user programming. We will discuss this further in the future work section. An alternative prototype interface for creating situations is shown in Figure 57. It uses iconic representations of commonly used context to allow users to construct situations by dragging these icons onto a panel and setting properties or values for the context types.

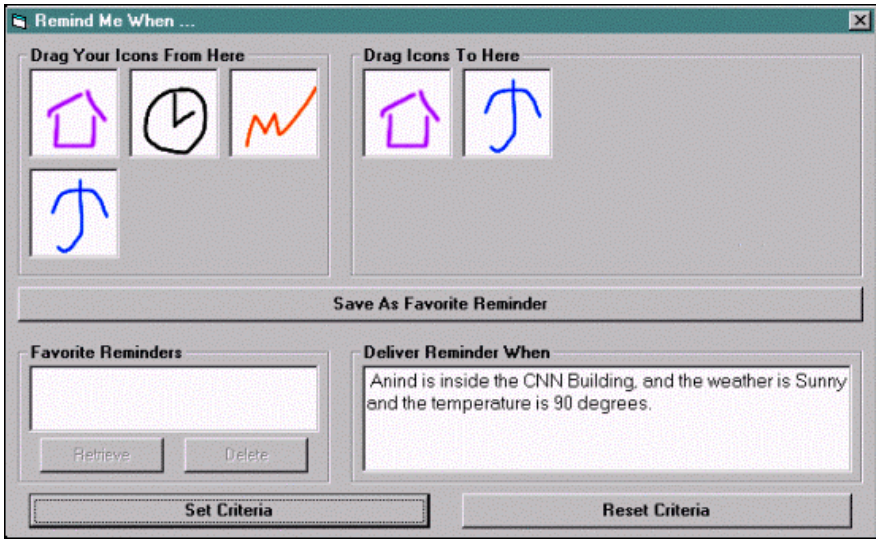


Figure 57: Alternative prototype for creating situations with icons.

Providing the end user with the ability to specify a situation was integral to this application. The use of the situation abstraction by the programmer was not. The application could have been written to use widgets, aggregators and interpreters directly, but instead of leveraging off the toolkit's ability to map between user-specified situations and these components, the application programmer would have to provide this ability making the application much more difficult to build.

The CybreMinder application also demonstrates the use of generalized context services. These are services that are supported by the architecture because they are thought to be useful by multiple applications. All the services used by the application have to do with communicating messages to users, whether by SMS, phone, pager, e-mail, or display on a nearby screen.

Finally, the CybreMinder application illustrated how applications could be prototyped in the absence of real sensors needed to sense necessary context information. For example, many sample situations that were prototyped in CybreMinder required the use of sensors that either had not been invented yet (*e.g.* determined user intention or feeling, similar to the level of interest indicator used in the Conference Assistant) or that we did not have access to (*e.g.* fine-grained positioning). Graphical user interfaces were used to simulate sensors that provided data to widgets. When real sensors are available, the widgets that

used the software sensors could simply be replaced with no change to the application. This indicates that this type of setup may also be useful for testing and prototyping application ideas on a desktop before deploying in the environment them with real sensors.

6.3.4 Situation Abstraction Summary

The situation abstraction allows application designers to program at a higher level and alleviates the designer from having to know about specific context components. It allows designers to treat the infrastructure as a single component and not have to deal with the details of individual components. In particular, this supports the ability to specify context requirements that bridge multiple components. This includes requirements for unrelated context that is acquired by multiple widgets and aggregators. It also includes requirements for interpreted context that is acquired by automatically connecting an interpreter to a widget or aggregator. Simply put, the situation abstraction allows an application designer to simply describe the context she wants and the situation she wants it in and to have the context infrastructure provide it. This power comes at the expense of additional abstraction. When designers do not want to know the details of context sensing, the situation abstraction is ideal. However, if the designer wants greater control over how the application acquires context from the infrastructure or wants to know more about the components in the infrastructure, the context component abstraction may be more appropriate. Note that the situation abstraction could not be supported without context components. The widgets, interpreters and aggregators with their uniform interfaces and ability to describe themselves to other components makes the situation abstraction possible.

It would have been easier to build all of the applications discussed in CHAPTER 5 with the situation abstraction, had it been available. The specification of what context was required in each application would have been simpler than using the Discoverer to locate relevant context components and to subscribe, query or request interpretation from each component individually. The In/Out Board, the Context-Aware Mailing List, DUMMBO, and the Conference Assistant would also have been enhanced by using the situation abstraction. All of these applications would be more resilient to context components being removed, would adapt better to the addition of new components and would be easier to evolve.

The situation abstraction is similar to a feature provided in stick-e notes (Brown 1996b) (section 3.4.2.1). Where stick-e notes allowed non-programmers to author a set of simple situations and to be presented with a specified message when one of those situations was realized. The stick-e notes infrastructure supports only a single application at a time. Stick-e notes used a centralized mechanism for determining what the current context was and comparing it to the authored situations. There was no support for acquiring context from sensors, storing context, interpreting it or making it available to any component other than this centralized comparison mechanism. In contrast, the situation abstraction allows both non-programmers and programmers to specify situations. The CybreMinder application is an example of allowing non-programmers to specify arbitrarily complex situations and to have a specified action take place when the situation is realized. Programmers are supported in using the situation abstraction in both new and existing applications without much overhead, through the use of BaseObject. The situation abstraction relies on the context component abstraction that supports the acquisition of context from sensors, interpretations and the distribution of context to many components.

6.4 Summary of the Investigations

In this chapter, we presented some further investigations in the field of context-aware computing that were facilitated by the Context Toolkit. We presented our explorations into the issues of mediating access to context in order to protect users' privacy and dealing with ambiguous context.

In the case of mediating access to context, we showed how the Context Toolkit could be extended to limit access to acquired context. By associating an owner with each piece of acquired context and allowing owners to specify rules about who can access their context and when, the infrastructure can determine what

context, if any, can be delivered to a requesting application or component. The Dynamic Door Display was used to demonstrate how this modified architecture could be used.

In the case of ambiguous context, we showed how the Context Toolkit could be combined with another architecture, OOPS, to allow users to be notified when context was ambiguous or uncertain. By providing notification before any action is taken, users are given the opportunity to make any corrections before a potentially irreversible action is taken. A modified In/Out Board was used to demonstrate the modified architecture built in this exploration.

We also introduced the situation abstraction for building context-aware applications. It simplifies the building and evolution of applications by not requiring designers to deal with the details of individual components, but rather deal with the entire context-sensing infrastructure as a whole. It allows a declarative style of programming where designers simply declare the context their applications require and the circumstances they require it in. It is the infrastructure's job to either deliver the requested context or indicate that it cannot deliver it. The use of the situation abstraction was demonstrated via the CybreMinder application.

In general, the Context Toolkit facilitates the building of context-aware applications. It makes them easy to design, build and evolve. Now that these applications are easier to build than before, we have the opportunity to investigate some of the more difficult issues in context-aware computing. In this chapter, we have shown that the Context Toolkit can be used as a research vehicle for investigating these issues.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis dissertation has described a conceptual framework and architecture that supports the designing, building, execution, and evolution of context-aware applications. In this final chapter, we summarize our research and briefly describe some areas that merit future research.

7.1 Research Summary

In CHAPTER 1, we presented definitions of context and context-awareness. We described the value of context in increasing the conversational bandwidth between humans, computational entities and the environment. CHAPTER 2 provided a summary of the related work in the field of context-aware computing. The related work ranged from applications that used no supporting abstractions for building applications to ones that used partial support. From this study of the related work, we identified three problems that make context-aware applications difficult to develop and that were not addressed by existing solutions:

1. a lack of variety of sensors for a given context type;
2. a lack of variety of context types; and,
3. an inability to evolve applications.

In CHAPTER 3, we introduced the design process for building context-aware applications. We showed that we could reduce and simplify the design process to the following steps:

1. *Specification*: Specify the problem being addressed and a high-level solution.
   1.1. Specify the context-aware behaviors to implement.
   1.2. Determine what context is required for these behaviors (with a knowledge of what is available from the environment) and request it.
2. *Acquisition*: Determine what hardware or sensors are available to provide that context and install them.
3. *Action*: Choose and perform context-aware behavior.

In addition, we presented the requirements for an architecture that supports this design process and addresses the three problems listed above. The requirements are:

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

We discussed relevant existing and proposed architectures and showed that none of these fully supported these requirements.

Further, we described how a framework that supports these requirements supports the simplified design process and makes it easier to build and evolve context-aware applications. We introduced the context component programming abstraction that uses context widgets, context interpreters, context aggregators and context services as building blocks.

In CHAPTER 4, we discussed our implementation of our supporting conceptual framework, the Context Toolkit, and how each requirement is supported. In CHAPTER 5, we discussed how the Context Toolkit was used to build a variety of applications, including the In/Out Board, Context-Aware Mailing List, DUMMBO, Intercom and the Conference Assistant. These applications demonstrated reuse of existing context components, the addition of context to a previously non-context-aware application, and the use of a large variety of context and sensors. Finally, in CHAPTER 6, we described how the Context Toolkit can be used as a research testbed to investigate some challenging issues in context-aware computing. We introduced the situation abstraction that programmers and end users can use to describe rich situations and be notified when they occur. This allows them to deal with the context-sensing infrastructure as a whole and not worry about the individual context components, simplifying the design process. It provides additional support for the use of new components, dealing with failed components, and evolving applications. We also discussed how extensions to the Context Toolkit have allowed us to investigate the issues of ambiguous context data and controlling access to context. By making it easier to build context-aware applications, we have been able to further investigate interesting issues in context-aware computing.

The contributions of this thesis are:

- the identification of a novel design process for building context-aware applications;
- the identification of architectural requirements to support the building and evolution of context-aware applications, resulting in an architecture that both "lowers the floor" (*i.e.* makes it easier for designers to build applications) and "raises the ceiling" (*i.e.* increases the ability of designers to build more sophisticated applications) in terms of providing this support;
- two programming abstractions to facilitate the design of context-aware applications: the context component abstraction and the situation abstraction;
- the exploration of the design space of context-aware computing through the development of a number of context-aware application that use a variety of context and support a variety of context-aware features; and,
- the building of the Context Toolkit that supports the above requirements, design process and programming abstractions, allowing us and others to use it as a research testbed to investigate new problems in context-aware computing, including controlling access to context, the mediation of ambiguous context and high-level programming abstractions.

7.2 Future Research Directions

The field of context-aware computing has a large scope and we obviously have not addressed all the relevant issues. We will now discuss some issues that warrant further investigation.

7.2.1 Context Descriptions

One of the problems that we have identified with both the situation and context component programming abstractions is the need for application programmers to use the same terminology as the context-sensing architecture. While this is not a problem in and of itself, the terminology used in the architecture was developed in an *ad hoc* manner. By terminology, we mean the names of context attributes, callbacks, widgets, interpreters and services. For a programmer to use a Location context widget, it must know that it has three attributes, a location, an identity and a timestamp. While the programmer can discover this at runtime, a logical categorization of the attributes and callbacks for each type of widget, interpreter and service would ease the development of applications. An interesting future research direction would be to develop a schema for describing context that would provide some structure to developing the terminology. This effort should not become the AI-complete problem of trying to model the entire world, but rather

should follow the methodology used in biology when developing schemas and ontologies to describe living things. The biological methodology provides structure while allowing constant evolution as new entities are discovered and added.

7.2.2 Prototyping Environment

The Context Toolkit facilitates the prototyping of applications despite the lack of available physical sensors, as discussed in 4.2.3.3. Programmers can use software sensors in the place of real physical sensors to build applications. When the real physical sensors become available and appropriate widgets are built, the application requires no change to make use of them. An interesting step would be to provide a richer prototyping environment. In this richer environment, programmers would not need to create simulated sensors but simply specify the context attributes their application requires and the data type for each. The environment could automatically create either a single simulated widget or multiple simulated widgets that allow programmers to prototype and test their applications and/or use currently executing widgets. The specification could be similar to that used in User Interface Management Systems (UIMS) (Olsen 1992), to automatically generate user interfaces from application code or higher-level specifications.

7.2.3 Sophisticated Interpreters

Our use of context interpreters in the applications we have built has generally been limited to simple converters that convert one type of context into another (*e.g.* iButton id to a username). The most sophisticated interpreter we built was the one used in the exploration of ambiguity that used motion detector data and historical information to infer who was entering or leaving the house. Context-aware computing has a lot to gain from the use of more sophisticated interpretation. This type of interpretation provides higher-level context information that cannot be readily sensed in a direct manner (*e.g.* a person is eating dinner). Higher-level context information is the key to producing more sophisticated context-aware applications.

Complex interpretation includes the use of context or sensor fusion, both when context widgets are providing the same type of information (homogeneous fusion when two components are providing the same type of information but are conflicting) or when widgets are providing different types of information (heterogeneous fusion). Learning techniques such as neural networks, classification techniques such as Hidden Markov Modeling (Rabiner and Juang 1986) and sensor fusion techniques (Brooks and Iyengar 1997) can and should be used to provide this interpretation. These techniques require that relevant context be acquired and stored, which the Context Toolkit already provides.

7.2.4 Composite Events

Currently with the Context Toolkit, application designers can construct situations from individual clauses connected by "AND" operators. This should be extended to include both OR and NOT operators. In addition, designers should be able to specify more complex composite events or situations (Nelson 1998), including ones that use the follow-by, disjunction, and without operators. The *follow-by* operator lets a programmer specify that one event must follow another event in time. The *disjunction* operator allows a programmer to specify an exclusive-or relationship between events. The *without* operator allows a programmer to specify that one event must follow another event without an intervening third event. By supporting these other operators, designers will be able to create much more complex situations than previously possible with the Context Toolkit.

7.2.5 Model of the Environment

One aspect that is missing from the current architecture is static knowledge about the environment. In the applications described in this thesis, there was an implied physical model (*e.g.* Building A is a building known to applications and components for the example In/Out Board discussed in 4.1). A model of the environment that was available to both the architecture and applications would enable both to perform more sophisticated inferencing. For example, let us assume that multiple sensors are used to determine a user's

location, but they are reporting different locations for a particular user. A knowledge of the physical space the user was moving through would aid in determining if any of the reported locations are invalid due to the time required to move from the last reported location to the current one (Schilit and Theimer 1994).

Along with a model of the physical space and the hierarchy it entails (…, city, street, building ,floor, room, …) (Brumitt, Shafer *et al.* 1999), a similar model of people and objects would be useful. A model of people would describe relationships between various people, including both work-related and social. A model of objects would also describe relationships between objects and their relationships to people and locations. In addition, commonsense knowledge of the world, in general, and the way things work would be beneficial to both the architecture and applications (CyCorp 2000; MindPixel 2000; Signiform 2000; Singh 2000). If the knowledge is placed in the architecture, all context-aware applications could query it for information relevant to the application's particular domain.

7.2.6 Quality of Service

One of the extensions to the Context Toolkit, which we described in the previous chapter, dealt with ambiguous context data. Ambiguity or accuracy is a type of quality of service metric. Other metrics include reliability, coverage, resolution, frequency and timeliness. The Context Toolkit should be extended to deal with all these metrics.

Reliability defines how tolerant the application is with regard to sensor failures. Usually, application designers will require high reliability. The Context Toolkit deals with reliability by recognizing when components become available or are no longer available, but expects that the sensors can detect their own failures or that Discoverers can. Coverage and resolution are related notions. *Coverage*, as described earlier in 6.3.2, defines the set of all possible values for a context attribute, and *resolution* defines the actual change that is required for the context attribute to change. These notions are easily illustrated with location. Suppose we wish to write an application that locates people in a home (as in the Communications Assistant). The required coverage for the location of people is the entire house. A resolution level of "room" is sufficient. Suppose an application is designed to turn on a light whenever people are within 6 feet and to dim it when people are within 9 feet. We need a resolution of at least 3 feet and, at a minimum, a non-contiguous coverage area that includes all 9 feet-radius spheres around fixtures.

Frequency and timeliness determine the "real-time" requirements of the application. *Frequency* defines how often the information needs to be updated. For example, if the application is a driving assistant that helps locate the nearest gas station, it may not make sense to update the location of the vehicle every second. An update every few minutes or under certain conditions detected by the application or other context-handling components (gas tank on empty, children hungry) is sufficient. *Timeliness* defines the time the application allows between the actual context change and the related notification to the application. If a person comes close to a light fixture, the light must turn on immediately. However, in the case of the application that locates people in the house, the application may allow a delay of a couple of minutes between the time a person comes home, and its identity is ascertained and known to the application. Once again, designers would ideally specify a zero timeliness but they must be ready to trade-off with the actual capabilities, and transmission and computation delays due to sensors and distribution.

Context components should indicate their baseline quality of service values for the context they can provide to applications and notify subscribers (and the Discoverer) of any changes to these values. This would allow application designers to specify their quality of service requirements as part of a specified situation, so the application could automatically switch to a different set of context components if the currently used set does not meet the specified requirements. This would require additional work of designers building context components and would require that a change be made to the situation algorithm laid out in Figure 50.

7.3 Conclusions

This thesis dissertation has presented the requirements for and an implementation of an architecture for supporting the designing, building, execution and evolution of context-aware applications. The architecture supports a simple design process for building applications, using either a context component programming model with widgets, interpreters, aggregators and services or a situation programming model. The architecture also allows the exploration of interesting research areas within the field of context-aware computing.

APPENDIX A

THE CONTEXT TOOLKIT

The Context Toolkit Web page is located at: http://www.cc.gatech.edu/fce/contexttoolkit
It contains:

- a short description of the Context Toolkit research;
- a list of Context Toolkit and context-related publications;
- a list of projects, including applications and components built both at Georgia Tech and elsewhere;
- a user's guide for the Context Toolkit, including an installation guide, tutorial and set of javadocs; and,
- information on how to download the publicly available Context Toolkit.

APPENDIX B

COMPONENTS AND APPLICATIONS

This appendix lists a number of the applications, widgets, interpreters, aggregators and services built, along with what languages they were built in, who built them and what sensors/actuators were used (if appropriate).

B-1 Applications

Table 6: List of applications developed with the Context Toolkit, the programming language used and who developed them.

Name	Language	Who Developed?
In/Out Board	Java	CTK developers
In/Out Board web version	PERL	CTK developers
In/Out Board web version	Frontier	CTK developers
In/Out Board WinCE version	Java	Other
Context-Aware Mailing List	Java	CTK developers
Information Display	Java	CTK developers
DUMMBO	Java	Other
InterCom	Java	Other
Digital Family Portrait	Squeak	Other
Smart Prototype Room*	Java	Other
Conference Assistant	Java	Both
Dynamic Door Display	Java	Other
CybreMinder	Java	CTK developers
CybreMinder with ambiguity	Java	Both
In/Out Board with ambiguity	Java	Both

* The Smart Prototype Room was a small environment in which other members of our research group explored the concept of smart environments, as a pre-cursor to the Aware Home (Kidd, Orr *et al.* 1999; Future Computing Environments 2000). Users were identified when they entered the room and their activities monitored as they interacted with appliances (television, stereo, video game console and lamps), furniture (windows and chairs) and read books and magazines. The Smart Prototype Room monitored these activities and controlled various aspects of the environment based on the user's context. For example, when a user took a magazine from a bookshelf, sat down in a recliner and reclined fully, the application determined that the user was relaxing so it turned on some relaxing music (via the stereo), altered the lighting and turned off the television.

B-2 Widgets

Table 7: List of widgets developed, the programming language used, who developed them and what sensor was used.

Name	Language	Who Developed?	Sensor
Location	Java	CTK developer	IButton
Location	Java	CTK developer	GUI
Location	Visual Basic	CTK developer	PinPoint
Location	Java	CTK developer	PinPoint
Location	Java	Other	WEST WIND
Location	Java	Both	Infrared motion detector
Beeper	Java	Other	GUI
Contact	Java	CTK developer	File
Display	Java	CTK developer	Display device
PhoneStatus	Java	Other	GUI
Group	Java	CTK developer	File
URL	Java	CTK developer	File
SoundLevel	Java	CTK developer	Microphone
SoundLevel	C++	Other	Microphone
SoundLevel	Java	Other	Microphone
Speech	Java	CTK developer	IBM ViaVoice
Temperature	Java	Other	Termperature sensor
Temperature	Java	Other	GUI
GPS	Java	Other	GPS unit
Weather	Java	Other	Web page
Floor	Java/C++	Other	Custom sensor
Memo	Java	Both	GUI
Registration	Java	CTK developer	GUI
PresentationContent	Java/C++	Both	PowerPoint & Internet Explorer
PresentationQuestion	Java	Both	GUI
Recording	Java	Both	Camera & microphones
Role	Java	Both	File & iButton
Calendar	Java	Both	File
Motion	Java/C++	Other	Infrared motion detector
DoorStatus	Java/C++	Other	Magnetic reed switch
TourRegistration	Java	CTK developer	GUI
TourDemo	Java	CTK developer	File & GUI
CabinetStatus	Java/C++	Other	Magnetic reed switch
Magazine	Java/C++	Other	RFID tags
ApplianceInUse	Java/C++	Other	Television, video games & lamps
Audio	Java/C++	Other	Microphone

B-3 Aggregators

Table 8: List of aggregators developed, the programming language used and who developed them.

Name	Language	Who Developed?
Location	Java	CTK developers
User	Java	CTK developers
Device	Java	CTK developers

B-4 Interpreters

Table 9: List of interpreters developed, the programming language used and who developed them.

Name	Language	Who Developed?
Demo Recommender	Java	CTK developers
Group to URL	Java	CTK developers
iButton ID to Group	Java	CTK developers
iButton ID to Group URL	Java	CTK developers
iButton ID to Name	Java	CTK developers
Conference Recommender	Java	CTK developers
Name to Presenter	Java	CTK developers
PinPoint Zone to Location	Java	CTK developers
PinPoint ID to Name	Java	CTK developers
Motion to Names	Java	CTK developers
Words to Valid Words	Java	CTK developers

B-5 Services

Table 10: List of services developed, the programming language used, who developed them and what actuator was used.

Name	Language	Who Developed?	Actuator
Display Choices	Java	CTK developers	Software
Display Message	Java	CTK developers	Software
Text to Speech	Java	Both	IBM ViaVoice
Beeper	Java	Other	GUI
Audio Switch	Java	Other	Audio Switch

APPENDIX C

XML CONTEXT TYPES AND MESSAGES

C-1 Context Types

Below is a list of all the context types that we have used in widgets and applications. This does not necessarily include context types that have been added by other researchers using the Context Toolkit.

```
iButtonID                 string
```
Represents the hexadecimal identity from an iButton

```
username                  string
```
Represents a user's name

```
group                     string
```
Represents a group that a user belongs to

```
url                       string
```
Represents a URL

```
userid                    string
```
Represents some abstract identity for a user

```
location                  string
```
Represents the location of a user or object

```
timestamp                 long
```
Represents the time at which an event occurred (number of milliseconds since January 1, 1970 00:00:00 GMT)

```
contactInfo               struct
  name                    string
  email                   string
  affiliation             string
```
Represents the contact information for a user, including the user's name, email address and affiliation

```
soundLevel                float
```
Represents the volume or sound level

```
interests                 string
```
Represents the interests of a user

```
interestLevel             string
```
Represents the interest level of a user in their activity

`inout` `string`
Represents the in or out status of a user in a location

`status` `string`
Represents some status information

`temperature` `float`
Represents the temperature in a location

`units` `string`
Represents the units used in a measurement

`width` `int`
Represents the width of some object

`height` `int`
Represents the height of some object

`graphics` `string`
Represents the graphics support of a display

`displayDevice` `string`
Represents the type of a display device

`demoName` `string`
Represents the name of a demonstration

`demoUrl` `string`
Represents the URL for a demonstration

`demoerUrl` `string`
Represents the URL for the giver of a demonstration

`keywords` `string`
Represents the keywords for some description

`description` `string`
Represents the description of some object or event

`demo` `string`
Represents a demonstration

`demos` `string`
Represents multiple demonstration

`demoInterest` `string`
Represents the interest in a demonstration

`demoInterests` `string`
Represents the cumulative interests in a demonstration

```
conferenceScheduleURL    string
```
Represents the URL for a conference schedule

```
phoneStatus              string
```
Represents the status for a phone

```
command                  string
```
Represents a command

```
data                     string
```
Represents some data

```
weatherType              string
```
Represents the weather for some location

```
sunriseTime              string
```
Represents the time of sunrise

```
sunsetTime               string
```
Represents the time of sunset

```
heartbeats               int
```
Represents the heart rate of a user

```
multipleAppointments     struct
  singleAppointment      struct
    startTime            long
    endTime              long
    description          string
```
Represents the appointments in a user's schedule

```
memoContent              struct
  presenter              string
  location               string
  topic                  string
  startTime              long
  endTime                long
  slideNumber            int
  interestLevel          string
  title                  string
  info                   string
  timestamp              long
  memoIndex              int
  imageURL               string
  memo                   string
```
Represents a user memo in the Conference Assistant

```
activityInfo             string
```
Represents the activity of a user

```
role                     string
```
Represents the role a user is playing or assuming

`doorState` `string`
Represents the state of a door

`motion` `string`
Represents the motion in a location

C-2 Context Messages

Below is a list of all the Context Toolkit context messages that are created by BaseObject, widgets, interpreters, aggregators and discoverers and the valid responses to those messages.

---

queryVersion

Message sent when a component wants to find out the version number of another component

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any application, widget, interpreter or aggregator

```
<queryVersion>
  <id> id_of_component_being_queried </id>
</queryVersion>
```

---

queryVersionReply

Reply sent when a component has been queried for its version number

Created in BaseObject by the queried application, widget, interpreter or aggregator
Sent to the requesting component

```
<queryVersionReply>
  <version> version_number_of_queried_component </version>
  <error> error_code </error>
</queryVersionReply>
```

Where `error_code` can be:

- `NoError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving component's id

---

pingComponent

Message sent when a component wants to ping another component to see if it is still available (still running)

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any application, widget, interpreter or aggregator

```
<ping>
  <id> id_of_component_being_pinged </id>
</ping>
```

---

pingComponentReply

Reply sent when a component has been pinged to see if it is available (still running)

Created in BaseObject by the ping'ed application, widget, interpreter or aggregator
Sent to the requesting component

```
<pingComponentReply>
  <error> error_code </error>
</pingComponentReply>
```

Where `error_code` can be:

`noError`, if there is no error

`invalidIdError`, if the received id does not match the receiving component's id

---

getWidgetAttributes

Message sent when a component wants to find out what the attributes for a particular widget or aggregator are

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<queryAttributes>
  <id> id_of_widget/aggregator_being_queried </id>
</queryAttributes>
```

---

getWidgetAttributesReply

Reply sent when a widget or aggregator has been queried for its context attributes

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<queryAttributesReply>
  <attributes>
    <attribute attributeType = attribute_type>
       name_of_attribute
    </attribute>
```

```
    …
  </attributes>
  <error> error_code </error>
</queryAttributesReply>
```

Where `attribute_type` can be `long`, `double`, `float`, `short`, `int`, `boolean`, `string` or `struct`. The default attribute_type is `string`, so if the attribute is a `string`, the "attribute type = attribute_type" phrase does not need to be included. If attribute_type is `struct`, then its value is a nested attributes object.

Where `error_code` can be:

`noError`, if there is no error

`invalidIdError`, if the received id does not match the receiving widget's or aggregator's id

`emptyResultError`, if there are no attributes for this widget

---

queryWidgetCallbacks

Message sent when a component wants to find out what the callbacks for a particular widget or aggregator are

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<queryCallbacks>
  <id> id_of_widget/aggregator_being_queried </id>
</queryCallbacks>
```

---

getWidgetCallbacksReply

Reply sent when a widget or aggregator has been queried for its callbacks

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<queryCallbacksReply>
  <callbacks>
    <callback>
      <callbackName> name_of_callback </callbackName>
      <attributes> … </attributes>
    </callback>
    …
  </callbacks>
  <error> error_code </error>
</queryCallbacksReply>
```

Where `error_code` can be:

`noError`, if there is no error

`invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
`emptyResultError`, if there are no callbacks for this widget

---

getWidgetServices

Message sent when a component wants to find out what the services for a particular widget or aggregator are

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<queryServices>
  <id> id_of_widget/aggregator_being_queried </id>
</queryServices>
```

---

getWidgetServicesReply

Reply sent when a widget or aggregator has been queried for its services

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<queryServicesReply>
  <services>
    <service>
      <serviceName> name_of_service </callbackName>
      <functions>
        <function>
          <functionName> name_of_function </functionName>
          <functionDescription>
            textual_description_of_function
          </functionDescription>
          <attributes> … </attributes>
          <functionTiming> timing_for_function
          </functionTiming>
        </function>
        …
      </functions>
    </service>
    …
  </services>
  <error> error_code </error>
</queryServicesReply>
```

Where `name_of_function` is the name of a particular function that this service provides

Where `timing_for_function` can be:
`synchronous`

asynchronous

Where error_code can be:
noError, if there is no error
invalidIdError, if the received id does not match the receiving widget's or aggregator's id
emptyResultError, if there are no services for this widget

---

subscribeTo

Message sent when a component wants to subscribe to a widget or aggregator

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<addSubscriber>
  <id> id_of_widget/aggregator_being_subscribed_to </id>
  <subscriber>
    <subscriberId> id_of_subscriber </subscriberId>
    <hostname> hostname_of_subscriber's_computer
    </hostname>
    <port> port_number_subscriber_receives_input_on </port>
    <callbackName> widget_callback_being_subscribed_to
    </callbackName>
    <callbackTag>
      tag_that_subscriber_uses_to_refer_to_callback
    </callbackTag>
    <conditions>
      <condition>
        <name> name_of_attribute_to_compare </name>
        <compare> operator_to_use_for_comparison </compare>
        <value> value_to_compare_to </value>
      </condition>
      …
    </conditions>
    <attributes> … </attributes>
  </subscriber>
</addSubscriber>
```

Where conditions are optional and are used to determine whether to send data to the subscriber

Where operator_to_use_for_comparison can be:
0, corresponds to =
1, corresponds to <=
2, corresponds to <
3, corresponds to >=
4, corresponds to >

Where `attributes` are optional and are the widget/aggregator attributes whose values are being requested. If the `name_of_attribute` in the first attribute is '`*`' or no attributes object is provided, then all data is returned

---

subscribeToReply

Reply sent when a widget or aggregator has been subscribed to

Created in Widget by the called widget or aggregator
Sent to the subscribing component

```
<addSubscriberReply>
  <error> error_code </error>
</addSubscriberReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `missingParameterError`, if the subscriber object passed in cannot be parsed
- `invalidAttributeError`, if one or more of the requested widget/aggregator attributes or condition attributes is not part of the widget/aggregator
- `invalidCallbackError`, if the callback being subscribed to is not part of the widget/aggregator

---

unsubscribeFrom

Message sent when a component wants to unsubscribe from a widget or aggregator

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<removeSubscriber>
  <id> id_of_widget/aggregator_being_subscribed_to </id>
  <subscriber> … </subscriber>
</removeSubscriber>
```

Where `conditions` and `attributes` in the subscriber are optional

---

unsubscribeFromReply

Reply sent when a widget or aggregator has been unsubscribed from

Created in Widget by the called widget or aggregator
Sent to the unsubscribing component

```
<removeSubscriberReply>
  <error> error_code </error>
</removeSubscriberReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `missingParameterError`, if the subscriber object passed in cannot be parsed
- `invalidCallbackError`, if the callback being unsubscribed from is not part of the widget/aggregator
- `unknownSubscriberError`, if the subscriber component is not a subscriber to the given widget/aggregator callback

---

subscriptionCallback

Message sent when a widget or aggregator wants to send callback data to a subscribing component

Created in Widget by any widget or aggregator sending data to their subscribers
Sent to any relevant subscribing component

```
<subscriptionCallback>
  <subscriberId> id_of_the_subscribing_component
  </subscriberId>
  <callbackTag>
    tag_used_to_reference_the_callback_by_the_subscriber
  </callbackTag>
  <attributeNameValues> … </attributeNameValues>
</subscriptionCallback>
```

---

subscriptionCallbackReply

Reply sent when a component receives callback information from a widget or aggregator

Created in BaseObject by the subscribing application, widget or aggregator
Sent to the widget or aggregator that originated the callback

```
<subscriptionCallbackReply>
  <subscriberId> id_of_the_subscribing_component
  </subscriberId>
  <data> … </data>
  <error> error_code </error>
</subscriptionCallbackReply>
```

Where `data` can contain any kind of data that the callback handler wants to return

Where `error_code` can be:

- `noError`, if there is no error

`missingParameterError`, if the `subscriber_id`, `subscriber_tag` or `attributeNameValues` object is missing

`unknownCallbackError`, if the `subscriber_tag` doesn't match the subscribing component's callback tag

`unknownSubscriberError`, if there is no handler registered to handle this callback

---

queryWidget

Message sent when a component wants to query a widget or aggregator for its latest context

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<query>
  <id> id_of_widget/aggregator_being_queried </id>
  <attributes> … </attributes>
</query>
```

Where `attributes` are optional and are the widget/aggregator attributes whose values are being requested. If the `name_of_attribute` in the first attribute is '`*`' or no attributes object is provided, then all data is returned

---

queryWidgetReply

Reply sent when a widget or aggregator has been queried for its latest context

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<queryReply>
  <attributeNameValues> … </attributeNameValues>
  <error> error_code </error>
</queryReply>
```

Where `error_code` can be:

`noError`, if there is no error

`invalidIdError`, if the received id does not match the receiving widget's or aggregator's id

`missingParameterError`, if the attributes object passed in cannot be parsed

`invalidAttributeError`, if one or more of the requested widget/aggregator attributes is not part of the widget/aggregator

`invalidDataError`, if no data is available to be returned

`incompleteDataError`, if not all the data can be returned

---

updateAndQueryWidget

Message sent when a component wants a widget or aggregator to acquire updated context information and return it

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<updateAndQuery>
  <id> id_of_widget/aggregator_being_queried </id>
  <attributes> … </attributes>
</updateAndQuery>
```

Where `attributes` are optional and are the widget/aggregator attributes whose values are being requested. If the `name_of_attribute` in the first attribute is '`*`' or no attributes object is provided, then all data is returned

---

updateAndQueryWidgetReply

Reply sent when a widget or aggregator has been told to update its context values and return them

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<updateAndQueryReply>
  <attributeNameValues> … </attributeNameValues>
  <error> error_code </error>
</updateAndQueryReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `missingParameterError`, if the attributes object passed in cannot be parsed
- `invalidAttributeError`, if one or more of the requested widget/aggregator attributes is not part of the widget/aggregator
- `invalidDataError`, if no data is available to be returned
- `incompleteDataError`, if not all the data can be returned

---

retrieveDataFrom

Message sent when a component wants to retrieve a widget or aggregator's historical context

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<retrieveData>
  <id> id_of_widget/aggregator_being_queried </id>
  <retrieval>
    <attributeFunctions>
      <attributeFunction attributeType = type_of_attribute,
                         function = type_of_function>
        name_of_attribute
      </attributeFunction>
      …
    </attributeFunctions>
    <conditions> … </conditions>
  </retrieval>
</retrieveData>
```

Where `type_of_function` can be:

- `none`, if no function is desired (makes `attributeFunction` behave like attribute)
- `max`, to return the maximum value for this attribute
- `min`, to return the minimum value for this attribute
- `count`, to return the number of values for this attribute
- `avg`, to return the average value for this attribute
- `sum`, to return the sum of all the values for this attribute

Where if the `name_of_attribute` in the first `attributeFunction` is '`*`', then all data (that satisfies all specified conditions) is returned

Where `conditions` are optional

---

retrieveDataFromReply

Reply sent when a widget or aggregator has been asked to return some historical context

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<retrieveDataReply>
  <retrievalResults>
    <attributeNameValues> … </attributeNameValues>
    …
  </retrievalResults>
  <error> error_code </error>
```

```
</retrieveDataReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `emptyResultError`, if there is no storage mechanism enabled or if the result set is empty
- `invalidRequestError`, if result set can not be parsed

---

putDataInWidget

Message sent when a component wants to put data into a widget or aggregator, simulating a callback

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<putData>
  <id> id_of_widget/aggregator_to_put_data_into </id>
  <callbackName> name_of_callback_to_simulate
  </callbackName>
  <attributeNameValues> … </attributeNameValues>
</putData>
```

---

putDataInWidgetReply

Reply sent when a widget or aggregator has had data put into it

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<putDataReply>
  <error> error_code </error>
</putDataReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `invalidDataError`, if the `attributeNameValues` object cannot be parsed or it contains no data
- `invalidCallbackError`, if the simulated callback is not part of the widget/aggregator
- `invalidAttributeError`, if the `attributeNameValues` object contains attributes that are not part of the widget/aggregator

---

runComponentMethod

Message sent when a component wants to execute another, non-standard method in a separate widget, aggregator or interpreter

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget, aggregator or interpreter

```
<name_of_method>
  <id> id_of_component_to_call_method_on </id>
  <attributeNameValues> … </attributeNameValues>
  <attributes> … </attributes>
</name_of_method>
```

---

runComponentMethod replies

Reply sent when a widget, aggregator or interpreter has been asked to execute some non-standard method

runWidgetMethod

Created in a particular widget/aggregator instance by the called widget or aggregator
Sent to the requesting component

```
<name_of_method_reply>
  <error> error_code </error>
</name_of_method_reply>
```

where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `unknownMethodError`, if the `name_of_method` does not match a widget/aggregator method and others specified by the widget/aggregator method

runInterpreterMethod

Created in a particular interpreter instance by the called interpreter
Sent to the requesting component

```
<name_of_method_reply>
  <error> error_code </error>
</name_of_method_reply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving interpreter's id

unknownMethodError, if the name_of_method does not match an interpreter method and others specified by the interpreter method

---

executeSynchronousService

Message sent when a component wants to execute a widget or aggregator's synchronous service

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<serviceRequest>
  <id> id_of_widget/aggregator_to_call_service_from </id>
  <functionTiming> synchronous </functionTiming>
  <serviceInput>
    <serviceName> name_of_service_to_execute </serviceName>
    <functionName> name_of_service_function_to_execute
    </functionName>
    <attributeNameValues> … </attributeNameValues>
  </serviceInput>
</serviceRequest>
```

---

executeSynchronousWidgetServiceReply

Reply sent when a widget or aggregator has been asked to execute a synchronous service

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<serviceRequestReply>
  <data>
    <status> service_status </status>
  </data>
  <error> error_code </error>
</serviceRequestReply>
```

Where data can contain any kind of data that the service wants to return

Where status can be:
- executed, if the service was successful
- failed, if the service failed

Where error_code can be:
- noError, if there is no error
- invalidIdError, if the received id does not match the receiving widget's or aggregator's id
- missingParameterError, if the given attributeNameValues are not sufficient for the service to execute
- unknownServiceError, if the service_name is not one of the widget's or aggregator's services

`unknownFunctionError`, if the `function_name` is not one of the service's functions

`invalidTimingError`, if the `function_timing` does not match the service function's timing

`invalidAttributeError`, if the `attributeNameValues` object contains attributes that are not part of the widget/aggregator's service function

---

executeAsynchronousWidgetService

Message sent when a component wants to execute a widget or aggregator's asynchronous service

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any widget or aggregator

```
<serviceRequest>
  <id> id_of_widget/aggregator_to_call_service_from </id>
  <functionTiming> asynchronous </functionTiming>
  <serviceInput>
    <serviceName> name_of_service_to_execute </serviceName>
    <functionName> name_of_service_function_to_execute
    </functionName>
    <serviceId>
      id_of_widget/aggregator_to_call_service_from
    </serviceId>
    <hostname> hostname_of_the_calling_component's_computer
    </hostname>
    <port>
      port_number_the_calling_component_receives_input_on
    </port>
    <id> id_of_the_calling_component </id>
    <requestTag>
      tag_that_component_uses_to_refer_to_service_reply
    </requestTag>
    <attributeNameValues> … </attributeNameValues>
  </serviceInput>
</serviceRequest>
```

---

executeAsynchronousWidgetServiceReply

Reply sent when a widget or aggregator has been asked to execute an asynchronous service

Created in Widget by the called widget or aggregator
Sent to the requesting component

```
<serviceRequestReply>
  <data>
    <status> service_status </status>
```

```
  </data>
  <error> error_code </error>
</serviceRequestReply>
```

Where `data` can contain any kind of data that the service wants to return

Where `status` can be:

- `executing`, if the service is successful so far
- `failed`, if the service failed

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving widget's or aggregator's id
- `missingParameterError`, if the given `attributeNameValues` are not sufficient for the service to execute
- `unknownServiceError`, if the `service_name` is not one of the widget's or aggregator's services
- `unknownFunctionError`, if the `function_name` is not one of the service's functions
- `invalidTimingError`, if the `function_timing` does not match the service function's timing
- `invalidAttributeError`, if the `attributeNameValues` object contains attributes that are not part of the widget/aggregator's service function

---

sendAsynchronousServiceResult

Message sent when an asynchronous service wants to return the service result to a requesting component

Created in Service by any asynchronous service sending data to a calling component
Sent to component that called the asynchronous service

```
<serviceResult>
  <id> id_of_the_calling_component </id>
  <serviceInput> … </serviceInput>
  <attributeNameValues> … </attributeNameValues>
</serviceResult>
```

---

sendAsynchronousServiceResultReply

Reply sent when a component receives the result of an asynchronous service execution

Created in BaseObject by the calling application, widget or aggregator
Sent to the widget or aggregator that originated the asynchronous service callback

```
<serviceResultReply>
  <data> … </data>
```

```
  <error> error_code </error>
</serviceResultReply>
```

Where data can contain any kind of data that the asynchronous callback handler wants to return

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving component's id
- `invalidRequestError`, if the given service tag does not match any of the receiving component's asynchronous callback tags
- `missingParameterError`, if there is an error in handling the asynchronous service callback
- `invalidRequestIdError`, if there is no handler registered to handle this callback

---

queryInterpreterInAttributes

Message sent when a component wants to find out what the input attributes for a particular interpreter are

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any interpreter

```
<queryInAttributes>
  <id> id_of_interpreter_being_queried </id>
</queryInAttributes>
```

---

queryInterpreterInAttributesReply

Reply sent when an interpreter has been queried for its input attributes

Created in Interpreter by the queried interpreter
Sent to the requesting component

```
<queryInAttributesReply>
  <attributes>
    <attribute attributeType = attribute_type>
      name_of_attribute
    </attribute>
    …
  </attributes>
  <error> error_code </error>
</queryInAttributesReply>
```

Where `attribute_type` can be `long`, `double`, `float`, `short`, `int`, `boolean`, `string` or `struct`. The default `attribute_type` is string, so if the attribute is a `string`, the "`attribute type = attribute_type`" phrase does not need to be included

If attribute_type is struct, then its value is a nested attributes object. For example:

```
<attributes>
  <attribute> date </attribute>
  <attribute attributeType = struct> contact information
  </attribute>
    <attributes>
      <attribute> name </attribute>
      <attribute> street address </attribute>
      <attribute> phone number </attribute>
      <attribute attribute_type = int> age </attribute>
    </attributes>
  </attribute>
</attributes>
```

Where error_code can be:

noError, if there is no error

invalidIdError, if the received id does not match the receiving interpreter's id

emptyResultError, if there are no incoming attributes for this interpreter

---

queryInterpreterOutAttributes

Message sent when a component wants to find out what the output attributes for a particular interpreter are

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any interpreter

```
<queryOutAttributes>
  <id> id_of_interpreter_being_queried </id>
</queryOutAttributes>
```

---

queryInterpreterOutAttributesReply

Reply sent when an interpreter has been queried for its input attributes

Created in Interpreter by the queried interpreter
Sent to the requesting component

```
<queryOutAttributesReply>
  <attributes>
    <attribute attributeType = attribute_type>
      name_of_attribute
    </attribute>
    …
  </attributes>
  <error> error_code </error>
```

```
</queryOutAttributesReply>
```

Where `attribute_type` can be `long`, `double`, `float`, `short`, `int`, `boolean`, `string` or `struct`. The default `attribute_type` is `string`, so if the attribute is a `string`, the "`attribute type = attribute_type`" phrase does not need to be included. If `attribute_type` is `struct`, then its value is a nested attributes object.

Where `error_code` can be:

`noError`, if there is no error

`invalidIdError`, if the received id does not match the receiving interpreter's id

`emptyResultError`, if there are no incoming attributes for this interpreter

---

askInterpreter

Message sent when a component wants an interpreter to interpret some context

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any interpreter

```
<interpret>
  <id> id_of_interpreter_to_request_interpretation_from
 </id>
  <attributeNameValues>
    <attributeNameValue attributeType = attribute_type>
      <attributeName> name_of_attribute </attributeName>
      <attributeValue> value_of_attribute </attributeValue>
   </attributeNameValue>
    …
  </attributeNameValues>
</queryOutAttributes>
```

Where `attribute_type` can be `long`, `double`, `float`, `short`, `int`, `boolean`, `string` or `struct`. The default `attribute_type` is `string`, so if the attribute is a `string`, the "`attribute type = attribute_type`" phrase does not need to be included. If `attribute_type` is `struct`, then its value is a nested `attributeNameValues` object, similar to a nested attributes object.

---

askInterpreterReply

Reply sent when an interpreter has finished interpreting context and returns it to the requesting component

Created in Interpreter by the called interpreter
Sent to the requesting component

```
<interpretReply>
  <attributeNameValues>
    <attributeNameValue attributeType = attribute_type>
```

```
      <attributeName> name_of_attribute </attributeName>
      <attributeValue> value_of_attribute </attributeValue>
   </attributeNameValue>
    …
  </attributeNameValues>
  <error> error_code </error>
</interpretReply>
```

Where `attribute_type` can be `long`, `double`, `float`, `short`, `int`, `boolean`, `string` or `struct`. The default `attribute_type` is `string`, so if the attribute is a `string`, the "`attribute type = attribute_type`" phrase does not need to be included. If `attribute_type` is `struct`, then its value is a nested `attributeNameValues` object, similar to a nested attributes object.

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received id does not match the receiving interpreter's id
- `missingParameterError`, if the provided attributes are not sufficient to perform interpretation
- `invalidAttributeError`, if one or more provided attributes cannot be used by the interpreter
- `invalidDataError`, if the interpretation fails for any other reason

---

addDiscoveredObject

Message sent when a widget, aggregator, or interpreter wants to register with a Discoverer

Created in BaseObject by any widget, interpreter or aggregator
Sent to any Discoverer

```
<addDiscoveredObject>
  <discovererId>      id_of_discoverer_being_registered_with
</discovererId>
  <component>
    <id> id_of_component_being_registered </id>
    <hostname> hostname_of_component_being_registered
    </hostname>
    <port>
      port_registering_component_receives_communications_on
    </port>
    <componentType> component_type </componentType>

    <interpreterData>
      <inAttributes> … </inAttributes>
      <outAttributes> … </outAttributes>
    <interpreterData>
```

or

```
    <widgetData>
      <attributes> … </attributes>
      <attributeNameValues> … </attributeNameValues>
      <callbacks> … </callbacks>
      <services> … </services>
    <widgetData>

  </component>
</addDiscoveredObject>
```

Where `component_type` can be `widget`, `aggregator` or `interpreter`

Where `inAttributes` and `outAttributes` are just attributes objects

Where only one of `interpreterData` or `widgetData` is required

---

addDiscoveredObjectReply

Reply sent when a Discoverer has been asked to register a widget, aggregator or interpreter

Created in Discoverer by the called discoverer
Sent to the registering widget, aggregator or interpreter

```
<addDiscoveredObjectReply>
  <error> error_code </error>
</addDiscoveredObjectReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received discoverer id does not match the receiving Discoverer's id
- `missingParameterError`, if one of the necessary parameters is missing
- `invalidDataError`, if the `widgetData` or `interpreterData` cannot be parsed

---

removeDiscoveredObject

Message sent when a widget, aggregator, or interpreter wants to unregister from a Discoverer

Created in BaseObject by any widget, interpreter or aggregator
Sent to any Discoverer

```
<removeDiscoveredObject>
  <discovererId> id_of_discoverer_being_unregistered_from
  </discovererId>
  <id> id_of_component_being_unregistered </id>
  <hostname> hostname_of_component_being_unregistered
  </hostname>
```

```
  <port>
    port_unregistering_component_receives_communications_on
  </port>
</removeDiscoveredObject>
```

---

removeDiscoveredObjectReply

Reply sent when a Discoverer is asked to unregister a widget, aggregator, or interpreter

Created in Discoverer by any discoverer
Sent to the unregistering widget, interpreter or aggregator

```
<removeDiscoveredObjectReply>
  <error> error_code </error>
</removeDiscoveredObjectReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError`, if the received discoverer id does not match the receiving Discoverer's id
- `invalidDataError`, if the component was not registered with the Discoverer

---

subscribeToDiscoveredObjects

Message sent when a component wants to subscribe to be notified when components register or unregister from a Discoverer

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any Discoverer

```
<subscribeToDiscoveredObjects>
  <discovererId> id_of_discoverer_being_registered_with
  </discovererId>
  <component> … </component>
  <subscriber>
    <subscriberId> id_of_subscriber </subscriberId>
    <hostname> hostname_of_subscriber's_computer
    </hostname>
    <port> port_number_subscriber_receives_input_on </port>
    <callbackTag>
      tag_that_subscriber_uses_to_refer_to_callback
    </callbackTag>
  </subscriber>
</subscribeToDiscoveredObjects>
```

Where `component` can contain any subset of component information: `hostname`, `port`, `id`, `component type`, and `input` and `output attributes` for interpreters or `attributes`, `constant attributes`, `callbacks` and `services` for widgets and aggregators

subscribeToDiscoveredObjectsReply

Reply sent when a Discoverer has been subscribed to

Created in Discoverer by the called Discoverer
Sent to the subscribing component

```
<subscribeToDiscoveredObjectsReply>
  <error> error_code </error>
</subscribeToDiscoveredObjectsReply>
```

Where `error_code` can be:

- `noError`, if there is no error
- `invalidIdError` if the received id does not match the receiving discoverer's id
- `missingParameterError` if the subscriber object passed in cannot be parsed
- `invalidDataError` if the component object cannot be parsed

---

unsubscribeFromDiscoveredObjects

Message sent when a component wants to unsubscribe from a Discoverer

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any Discoverer

```
<removeDiscovererSubscriber>
  <id> id_of_discoverer_being_subscribed_to </id>
  <subscriber> … </subscriber>
</removeDiscovererSubscriber>
```

Where `conditions` and `attributes` in the subscriber are optional

---

unsubscribeFromDiscoveredObjectsReply

Reply sent when a Discoverer has been unsubscribed from

Created in Discoverer by the called discoverer
Sent to the unsubscribing component

```
<removeDiscovererSubscriberReply>
  <error> error_code </error>
</removeDiscovererSubscriberReply>
```

Where `error_code` can be:

- `noError`, if there is no error

`invalidIdError`, if the received id does not match the receiving discoverer's id

`missingParameterError`, if the subscriber object passed in cannot be parsed

`unknownSubscriberError`, if the subscriber component is not a subscriber to the Discoverer

---

discovererCallback

Message sent when a Discoverer sends information about new components that have registered with it, to subscribers

Created in Discoverer by a discoverer
Sent to any relevant subscribing component

```
<discovererCallback>
  <id> id_of_the_subscribing_component </id>
  <callbackTag>
    tag_used_to_reference_the_callback_by_the_subscriber
  </callbackTag>
  <components>
    <component> … </component>
    <componentStatus> component_status </componentStatus>
    …
  </components>
</discovererCallback>
```

Where `component_status` can be:

`registered`, if the corresponding component registered with the Discoverer

`unregistered`, if the corresponding component unregistered from the Discoverer

---

discovererCallbackReply

Reply sent when a Discoverer has updated a component about newly discovered components

Created in BaseObject by the subscribing component
Sent to the Discoverer that sent the callback

```
<discovererCallbackReply>
  <error> error_code </error>
</discovererCallbackReply>
```

Where `error_code` can be:

`noError`, if there is no error

`missingParameterError`, if the `subscriber_id`, `subscriber_tag` or any `component data` is missing

unknownCallbackError, if the callbackTag doesn't match the subscribing component's callback tag

unknownSubscriberError, if there is no handler registered to handle this callback

---

retrieveKnownDiscoveredObjects

Message sent when a component wants to retrieve components registered with a Discoverer

Created in BaseObject by any application, widget, interpreter or aggregator
Sent to any Discoverer

```
<retrieveKnownDiscoveredObjects>
  <discovererId> id_of_discoverer_being_registered_with
  </discovererId>
  <component> … </component>
</retrieveKnownDiscoveredObjects>
```

Where component can contain any subset of component information: hostname, port, id, component type, and input and output attributes for interpreters or attributes, constant attributes, callbacks and services for widgets and aggregators

---

retrieveKnownDiscoveredObjectsReply

Reply sent when a Discoverer replies to a query about discovered objects

Created in Discoverer by the queried discoverer
Sent to the Discoverer that sent the callback

```
<discovererCallbackReply>
  <error> error_code </error>
</discovererCallbackReply>
```

Where error_code can be:

noError, if there is no error

missingParameterError, if the subscriber_id, subscriber_tag or any component data is missing

invalidDataError, if the component data cannot be parsed

BIBLIOGRAPHY

Abowd, Gregory D., Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper and Mike Pinkerton (1997). Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks* **3**(5): pp. 421-433. October 1997.
Available at: http://www.acm.org/pubs/articles/journals/wireless/1997-3-5/p421-abowd/p421-abowd.pdf
(cited on pp. 4, 12, 14, 21)

Abowd, Gregory D., Anind K. Dey, Robert J. Orr and Jason Brotherton (1998). Context-awareness in wearable and ubiquitous computing. *Virtual Reality* **3**: pp. 200-211. 1998.
Available at: http://www.cc.gatech.edu/fce/ctk/pubs/VRSIJ-3.pdf
(cited on pp. 5, 12, 22)

Adly, Noha, Pete Steggles and Andy Harter (1997). SPIRIT: A resource database for mobile users. In the *Workshop on Ubiquitous Computing, affiliated with the ACM Conference on Human Factors in Computer Systems (CHI '97)*, Atlanta, GA. March 22-27, 1997.
Available at: ftp://ftp.uk.research.att.com/pub/docs/att/paper.97.2.ps.Z
(cited on p. 20)

Arons, Barry (1991). The design of audio servers and toolkits for supporting speech in the user interface. *Journal of the American Voice I/O Society* **9**: pp. 27-41. March 1991.
Available at: http://www.media.mit.edu/~barons/aviosj91.html
(cited on p. 27)

Bauer, Martin, Timo Heiber, Gerd Kortuem and Zary Segall (1998). A collaborative wearable system with remote sensing. In the *Proceedings of the 2nd IEEE International Symposium on Wearable Computers (ISWC'98)*, pp. 10-17, Pittsburgh, PA, IEEE. October 19-20, 1998.
Available at: http://www.cs.uoregon.edu/research/wearables/Papers/ISWC98-bauer.ps
(cited on pp. 20, 28, 28)

Bederson, Ben B. (1995). Audio augmented reality: A prototype automated tour guide. In the *Proceedings of the 1995 ACM Conference on Human Factors in Computing Systems (CHI '95)*, pp. 210-211, Denver, CO, ACM. May 7-11, 1995.
Available at: http://www.acm.org/sigs/sigchi/chi95/Electronic/documnts/shortppr/bbb_bdy.htm
(cited on p. 14)

Berners-Lee, Tim, Roy T. Fielding and Henrik F. Nielsen (1996). Hypertext Transfer Protocol HTTP/1.0. Web page.
Available at: http://www.ietf.org/rfc/rfc1945.txt?number=1945
(cited on p. 46)

Box, Don, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte and Dave Winer (2000). Simple Object Access Protocol (SOAP) 1.1. Web page.
Available at: http://www.w3.org/TR/SOAP/
(cited on p. 67)

Bray, Tim (2000). The Annotated XML Specification. Web page.
Available at: http://www.xml.com/axml/testaxml.htm
(cited on p. 46)

Brooks, Frederick P. (1987). No silver bullet; Essence and accidents of software engineering. *IEEE Computer* **20**(4): pp. 10-19. April 1987.
Available at: http://www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html
(cited on p. 23)

Brooks, Richard R. and Sundararaja S. Iyengar (1997). *Multi-sensor fusion: Fundamentals and applications with software*. 1st edition. Prentice Hall, Englewood Cliffs, NJ.
(cited on pp. 93, 122)

Brotherton, Jason, Gregory D. Abowd and Khai Truong (1999). Supporting capture and access interfaces for informal and opportunistic meetings. Technical Report GIT-GVU-99-06. Georgia Institute of Technology, GVU Center. Atlanta, GA.
Available at: ftp://ftp.cc.gatech.edu/pub/gvu/tr/1999/99-06.pdf
(cited on p. 74)

Brown, Martin G. (1996a). Supporting user mobility. In the *Proceedings of the IFIP Conference on Mobile Communications (IFIP'96)*, Canberra, Australia, IFIP. September 1996.
Available at: ftp://ftp.uk.research.att.com/pub/docs/att/paper.96.7.pdf
(cited on pp. 5, 12)

Brown, Peter J. (1996b). The Stick-e Document: A framework for creating context-aware applications. In the *Proceedings of the Electronic Publishing '96*, pp. 259-272, Laxenburg, Austria, IFIP. September 1996.
Available at: http://www.cs.ukc.ac.uk/research/infosys/mobicomp/Fieldwork/Papers/ps/StickeDocument.ps
(cited on pp. 3, 12, 31, 118)

Brown, Peter J. (1998). Triggering information by context. *Personal Technologies* **2**(1): pp. 1-9. March 1998.
Available at: http://www.cs.ukc.ac.uk/people/staff/pjb/papers/personal_technologies.htm
(cited on pp. 5, 12)

Brown, Peter J., John D. Bovey and Xian Chen (1997). Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications* **4**(5): pp. 58-64. October 1997.
Available at: http://www.cs.ukc.ac.uk/people/staff/pjb/papers/personal_comms.html
(cited on pp. 3, 5, 12, 32)

Brumitt, Barry L., Brian Meyers, Jon Krumm, Amanda Kern and Steve Shafer (2000). EasyLiving: Technologies for Intelligent Environments. In the *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pp. 12-27, Bristol, UK, Springer-Verlag. September 25-27, 2000.
Available at: http://www.research.microsoft.com/barry/research/huc2k-final.pdf
(cited on p. 35)

Brumitt, Barry L., Steve Shafer, John Krumm and Brian Meyers (1999). EasyLiving and the role of geometry in ubiquitous computing. In the *DARPA/NIST/NSF Workshop on Research Issues in Smart Computing Environments*, Atlanta, GA. July 25, 1999.
Available at: http://www.research.microsoft.com/barry/research/ELandGeometry2.pdf
(cited on pp. 35, 123)

Butz, Andreas, Clifford Beshers and Steven Feiner (1998). Of vampire mirrors and privacy lamps: Privacy management in multi-user augmented environments. In the *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST '98)*, pp. 171-172, San Francisco, CA, ACM. November 2-4, 1998.
Available at: http://www.cs.columbia.edu/~butz/publications/papers/uist98.pdf
(cited on p. 93)

Caswell, Deborah and Phillippe Debaty (2000). Creating Web representations for places. In the *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pp. 114-126, Bristol, UK, Springer-Verlag. September 25-27, 2000.
Available at: http://www.cooltown.hp.com/papers/placeman/PlaceManagerPublic.doc
(cited on p. 32)

Cheyer, Adam J. and Luc Julia (1995). Multimodal Maps: An agent-based approach. In the *Proceedings of the International Conference on Cooperative Multimodal Communication (CMC '95)*, pp. 103-113, Eindhoven, The Netherlands. May 24-26, 1995.
Available at: ftp://ftp.ai.sri.com/pub/papers/cheyer-cmc95.ps.gz
(cited on p. 93)

Clark, Herbert H. and Susan E. Brennan (1991). *Grounding in communication*. Perspectives on Socially Shared Cognition. pp. 127 - 149. L. Resnick, J. Levine and S. Teasley, Editors. American Psychological Society, Washington, DC.
(cited on p. 2)

Coen, Michael, Brenton Philips, Nimrod Warshawshy and Luke Weisman (1999). Meeting the computational needs of intelligent environments: the MetaGlue environment. In the *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)*, pp. 201-212, Dublin, Ireland, Springer-Verlag. December 13-14, 1999.
Available at: http://www.ai.mit.edu/people/mhcoen/metaglue.pdf
(cited on p. 31)

Cohen, Philip R., Adam J. Cheyer, Michelle Wang and Soon C. Baeg (1994). An Open Agent Architecture. In the *Proceedings of the AAAI Spring Symposium Series on Software Agents (AAAI Technical Report SS-94-03)*, pp. 1-8, Palo Alto, CA, AAAI. March 21-23, 1994.
Available at: http://www.ai.sri.com/pubs/papers/Cohe9403-1:Open/document.ps.gz
(cited on pp. 30, 68)

Cooperstock, Jeremy R., Koichiro Tanikoshi, Garry Beirne, Tracy Narine and William Buxton (1995). Evolution of a reactive environment. In the *Proceedings of the 1995 ACM Conference on Human Factors in Computing Systems (CHI '95)*, pp. 170-177, Denver, CO, ACM. May 7-11, 1995.
Available at: http://www1.acm.org/sigs/sigchi/chi95/Electronic/documnts/papers/jrc_bdy.htm
(cited on pp. 5, 12, 16)

Covington, Michael J., Wende Long, Srividhya Srinivasan, Anind K. Dey, Mustaque Ahamad and Gregory D. Abowd (2001). Securing context-aware applications using environment roles. In submission to the *6th ACM Symposium on Access Control Models and Technologies (SACMAT 2001)*, Chantilly, VA, ACM. May 3-4, 2001.
Available at: ftp://ftp.cc.gatech.edu/pub/coc/tech_reports/2000/GIT-CC-00-29.ps.Z
(cited on p. 92)

Covington, Michael J., Matthew J. Moyer and Mustaque Ahamad (2000). Generalized role-based access control for securing future applications. In the *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, MD, NIST. October 16-19, 2000.
Available at: http://www.cc.gatech.edu/fce/house/research/pubs/GIT-CC-00-02.pdf
(cited on p. 92)

CyCorp (2000). Cyc Knowledge Server. Web page.
Available at: http://www.cyc.com/
(cited on p. 123)

Dallas Semiconductor (1999). iButton Home Page. Web page.
Available at: http://www.ibutton.com/
(cited on p. 22)

Davies, Nigel, Keith Mitchell, Keith Cheverst and Gordon Blair (1998). Developing a context-sensitive tour guide. In the *1st Workshop on Human Computer Interaction for Mobile Devices*, Glasgow, Scotland. May 21-23, 1998.
Available at: http://www.dcs.gla.ac.uk/~johnson/papers/mobile/HCIMD1.html#_Toc420818986
(cited on pp. 5, 12, 14)

Davies, Nigel, Stephen Wade, Adrian Friday and Gordon Blair (1997). Limbo: A tuple space based platform for adaptive mobile applications. In the *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pp. 291-302, Toronto, Canada. May 26-30, 1997.
Available at: ftp://ftp.comp.lancs.ac.uk/pub/mpg/MPG-97-02.ps.Z
(cited on pp. 12, 16, 68)

DeVaul, Richard W. and Alex Pentland (2000). The Ektara architecture: The right framework for context-aware wearable and ubiquitous computing applications. Massachusetts Institute of Technology. Cambridge, MA.
Available at: http://www.media.mit.edu/~rich/DPiswc00.pdf
(cited on p.37)

Dey, Anind K. (1998). Context-aware computing: The CyberDesk project. In the *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments (AAAI Technical Report SS-98-02)*, pp. 51-54, Palo Alto, CA, AAAI Press. March 23-25, 1998.
Available at: http://www.cc.gatech.edu/fce/cyberdesk/pubs/AAAI98/AAAI98.html
(cited on pp. 3, 5, 12, 33, 35)

Dey, Anind K. and Gregory D. Abowd (1997). CyberDesk: The use of perception in context-aware computing. In the *Proceedings of the 1997 Workshop on Perceptual User Interfaces (PUI '97)*, pp. 26-27, Banff, Alberta. October 19-21, 1997.
Available at: http://www.cc.gatech.edu/fce/cyberdesk/pubs/PUI97/pui.html
(cited on p. 5)

Dey, Anind K. and Gregory D. Abowd (2000a). CybreMinder: A context-aware system for supporting reminders. In the *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, pp. 172-186, Bristol, UK, Springer-Verlag. September 25-27, 2000.
Available at: http://www.cc.gatech.edu/fce/contexttoolkit/pubs/HUC2000.pdf
(cited on p. 110)

Dey, Anind K. and Gregory D. Abowd (2000b). Towards a better understanding of context and context-awareness. In the *Workshop on the What, Who, Where, When and How of Context-Awareness, affiliated with the 2000 ACM Conference on Human Factors in Computer Systems (CHI 2000)*, The Hague, Netherlands. April 1-6, 2000.
Available at: ftp://ftp.cc.gatech.edu/pub/gvu/tr/1999/99-22.pdf
(cited on pp. 20, 35)

Dey, Anind K., Gregory D. Abowd and Andrew Wood (1998). CyberDesk: A framework for providing self–integrating context–aware services. *Knowledge Based Systems* **11**(1): pp. 3-13. September 30, 1998.
Available at: http://www.cc.gatech.edu/fce/ctk/pubs/KBS11-1.pdf
(cited on pp. 3, 5, 12, 28, 33)

Dey, Anind K., Masayasu Futakawa, Daniel Salber and Gregory D. Abowd (1999). The Conference Assistant: Combining context-awareness with wearable computing. In the *Proceedings of the 3rd International Symposium on Wearable Computers (ISWC'99)*, pp. 21-28, San Francisco, CA, IEEE. October 20-21, 1999.
Available at: http://www.cc.gatech.edu/fce/ctk/pubs/ISWC99.pdf
(cited on p. 80)

Dey, Anind K., Daniel Salber and Gregory D. Abowd (1999). A context-based infrastructure for smart environments. In the *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)*, pp. 114-128, Dublin, Ireland, Springer Verlag. December 13-14, 1999.
Available at: http://www.cc.gatech.edu/fce/contexttoolkit/pubs/MANSE99.pdf
(cited on pp. 44, 68, 73)

Dey, Anind K., Daniel Salber and Gregory D. Abowd (2001). A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction* **16**. To appear in 2001.
Available at: http://www.cc.gatech.edu/fce/contexttoolkit/pubs/HCIJ16.pdf
(cited on pp. 19, 44)

Elrod, Scott, Gene Hall, Rick Costanza, Michael Dixon and Jim des Rivieres (1993). Responsive office environments. *Communications of the ACM* **36**(7): pp. 84-85. July 1993.
Available at: http://www.acm.org/pubs/articles/journals/cacm/1993-36-7/p84-elrod/p84-elrod.pdf
(cited on pp. 5, 12)

Everyday Computing Lab (2000). Everyday Computing Lab Projects. Web page.
Available at: http://www.cc.gatech.edu/~everyday-computing/projects.htm
(cited on p. 93)

Feiner, Steven, Blair MacIntyre, Tobias Hollerer and Anthony Webster (1997). A Touring Machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal Technologies* **1**(4): pp. 208-217. 1997.
Available at: http://www.cc.gatech.edu/fac/Blair.MacIntyre/papers/perstech.pdf
(cited on p. 14)

Fels, Sidney, Yasuyuki Sumi, Tameyuki Etani, Nicolas Simonet, Kaoru Kobayshi and Kenji Mase (1998). Progress of C-MAP: A context-aware mobile assistant. In the *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments (AAAI Technical Report SS-98-02)*, pp. 60-67, Palo Alto, CA, AAAI Press. March 23-25, 1998.
Available at: http://www.mic.atr.co.jp/~fels/papers/AAAI.spring.symposium.1998.ps
(cited on p. 14)

Fickas, Stephen, Gerd Kortuem and Zary Segall (1997). Software organization for dynamic and adaptable wearable systems. In the *Proceedings of the 1st International Symposium on Wearable Computers (ISWC'97)*, pp. 56-63, Cambridge, MA, IEEE. October 13-14, 1997.
Available at: http://www.cs.uoregon.edu/research/wearables/Papers/iswc97.ps
(cited on pp. 5, 5, 12, 17)

Franklin, David and Joshua Flaschbart (1998). All gadget and no representation makes jack a dull environment. In the *Proceedings of the AAAI 1998 Spring Symposium on Intelligent Environments (AAAI Technical Report SS-98-02)*, pp. 155-160, Palo Alto, CA, AAAI Press. March 23-25, 1998.
Available at: http://www.ils.nwu.edu/~franklin/Pubs/Franklin-SprSym98.pdf
(cited on p. 3)

Friday, Adrian (1996). Infrastructure support for adaptive mobile applications. PhD dissertation. Computing Department, University of Lancaster. Lancaster, UK.
Available at: ftp://ftp.comp.lancs.ac.uk/pub/mpg/MPG-96-40.ps.gz
(cited on p. 16)

Future Computing Environments (2000). Georgia Tech's Aware Home Research Initiative. Web page.
Available at: http://www.cc.gatech.edu/fce/ahri/
(cited on pp. 96, 126)

Harrison, Beverly L., Kenneth P. Fishkin, Anuj Gujar, Carlos Mochon and Roy Want (1998). Squeeze me, hold me, tilt me! An exploration of manipulative user interfaces. In the *Proceedings of the CHI'98 Conference on Human Factors in Computer Systems*, pp. 17-24, Los Angeles, CA, ACM. April 18-23, 1998.
Available at: http://www.parc.xerox.com/csl/members/want/papers/squeezy-chi-apr98.pdf
(cited on pp. 12, 13)

Harter, Andy and Andy Hopper (1994). A distributed location system for the Active Office. *IEEE Networks* **8**(1): pp. 62-70. January 1994.
Available at: ftp://ftp.uk.research.att.com/pub/docs/att/tr.94.1.ps.Z
(cited on p. 20)

Harter, Andy, Andy Hopper, Pete Steggles, Andy Ward and Paul Webster (1999). The anatomy of a context-aware application. In the *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom'99)*, pp. 59-68, Seattle, WA, ACM. August 15-19, 1999.
Available at: ftp://ftp.uk.research.att.com/pub/docs/att/tr.1999.7.pdf
(cited on p. 20)

Hewlett Packard (2000). Embedded Software from HP Chai. Web page.
Available at: http://www.embedded.hp.com/index.html
(cited on p. 67)

Hinckley, Ken, Jeff Pierce, Mike Sinclair and Eric Horvitz (2000). Sensing techniques for mobile interaction. In the *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST 2000)*, pp. 91-100, San Diego, CA, ACM. November 5-8, 2000.
Available at: http://research.microsoft.com/users/kenh/papers/PPC-Sensing_color.pdf
(cited on pp. 12, 13)

Hudson, Scott E. (1997). Principles of User Interface Software: Toolkits. Class notes. Georgia Institute of Technology, Atlanta, GA. March 1997.
(cited on p. 26)

Hull, Richard, Philip Neaves and James Bedford-Roberts (1997). Towards situated computing. In the *Proceedings of the 1st International Symposium on Wearable Computers (ISWC'97)*, pp. 146-153, Cambridge, MA, IEEE. October 13-14, 1997.
Available at: http://fog.hpl.external.hp.com/techreports/97/HPL-97-66.pdf
(cited on pp. 3, 5, 36)

IBM (2000). IBM Voice Systems home page. Web page.
Available at: http://www.ibm.com/software/speech/
(cited on p. 78)

Johnson, Jeff (1992). Selectors: Going beyond user-interface widgets. In the *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '92)*, pp. 273-279, Monterey, CA, ACM. May 3-7, 1992.
Available at: http://www.acm.org/pubs/articles/proceedings/chi/142750/p273-johnson/p273-johnson.pdf
(cited on p. 39)

Kidd, Cory D., Thomas O'Connell, Kris Nagel, Sameer Patil and Gregory D. Abowd (2000). Building a Better Intercom: Context-Mediated Communication within the Home. Technical Report Georgia Institute of Technology, GVU Center. Atlanta, GA.
(cited on p. 78)

Kidd, Cory D., Robert J. Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E. Starner and Wendy Newstetter. (1999). The Aware Home: A living laboratory for ubiquitous computing research. In the *Proceedings of the 2nd International Workshop on Cooperative Buildings (CoBuild '99)*, pp. 191-198, Pittsburgh, PA, Springer-Verlag. October 1-2, 1999.
Available at: http://www.cc.gatech.edu/fce/house/cobuild99_final.html
(cited on pp. 96, 126)

Kortuem, Gerd, Zary Segall and Martin Bauer (1998). Context-aware, adaptive wearable computers as remote interfaces to 'intelligent' environments. In the *Proceedings of the 2nd International Symposium on Wearable Computers (ISWC '98)*, pp. 58-65, Pittsburgh, PA, IEEE. October 19-20, 1998.
Available at: http://www.cs.uoregon.edu/research/wearables/Papers/ISWC98-kortuem.ps
(cited on pp. 5, 12, 17, 20)

Kortuem, Gerd, Zary Segall and T.G.C. Thompson (1999). Close encounters: Supporting mobile collaboration through interchange of user profiles. In the *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC '99)*, pp. 171-185, Karlsruhe, Germany, Springer-Verlag. September 27-29, 1999.
Available at: http://www.cs.uoregon.edu/research/wearables/Papers/kortuem-huc99.ps
(cited on p. 115)

Kumar, Sanjeev, Phil .R. Cohen and Hector J. Levesque (2000). The Adaptive Agent Architecture: Achieving fault-tolerance using persistent broker teams. In the *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS 2000)*, pp. 159-166, Boston, MA, IEEE. July 7-12, 2000.
Available at: http://cse.ogi.edu/CHCC/Papers/sanjeevPaper/icmas2000.pdf
(cited on p. 30)

Lamming, Mik and Mike Flynn (1994). Forget-me-not: Intimate computing in support of human memory. In the *Proceedings of the FRIEND 21: International Symposium on Next Generation Human Interfaces*, pp. 125-128, Meguro Gajoen, Japan. 1994.
Available at: http://www.xrce.xerox.com/publis/cam-trs/pdf/1994/epc-1994-103.pdf
(cited on pp. 8, 82)

Lau, Tessa A., Oren Etzioni and Daniel S. Weld (1999). Privacy interfaces for information management. *Communications of the ACM* **42**(10): pp. 88-94. October 1999.
Available at: ftp://ftp.cs.washington.edu/tr/1998/02/UW-CSE-98-02-01.PS.Z
(cited on p. 88)

Long, Sue, Rob Kooper, Gregory D. Abowd and Christopher G. Atkeson (1996). Rapid prototyping of mobile context-aware applications: The Cyberguide case study. In the *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96)*, pp. 97-107, White Plains, NY, ACM. November 10-12, 1996.
Available at: http://www.cc.gatech.edu/fce/cyberguide/pubs/mobicom96-cyberguide.ps
(cited on pp. 12, 14)

Lyons, Kent, Cory D. Kidd and Thad E. Starner (2000). Widespread easy and subtle tracking with wireless identification networkless devices -- WEST WIND: An environmental tracking system. Technical Report GIT-GVU-00-15. Georgia Institute of Technology, GVU Center. Atlanta, GA.
Available at: ftp://ftp.cc.gatech.edu/pub/gvu/tr/2000/00-15.pdf
(cited on p. 78)

MacIntyre, Blair and Steven Feiner (1996). Language-level support for exploratory programming of distributed virtual environments. In the *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST'96)*, pp. 83-94, Seattle, WA, ACM. November 6-8, 1996.
Available at: http://www.cc.gatech.edu/fac/Blair.MacIntyre/papers/uist96.pdf
(cited on p. 27)

Mankoff, Jennifer, Gregory D. Abowd and Scott E. Hudson (2000). OOPS: A toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers and Graphics* **24**(6). To appear in 2000.
Available at: http://www.cc.gatech.edu/fce/errata/publications/computers-and-graphics00.pdf
(cited on p. 94)

Mankoff, Jennifer, Scott E. Hudson and Gregory D. Abowd (2000a). Interaction techniques for ambiguity resolution in recognition-based interfaces. In the *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology (UIST 2000)*, pp. 11-20, San Diego, CA, ACM. November 5-8, 2000.
Available at: http://www.cc.gatech.edu/fce/errata/publications/uist-oops00.pdf
(cited on p. 94)

Mankoff, Jennifer, Scott E. Hudson and Gregory D. Abowd (2000b). Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. In the *Proceedings of the CHI 2000 Conference on Human Factors in Computer Systems*, pp. 368-375, The Hague, Netherlands, ACM. April 1-6, 2000.
Available at: http://www.cc.gatech.edu/fce/errata/publications/chi00.pdf
(cited on p. 94)

Mills, David (1996). Simple Network Time Protocol (SNTP) Version 4 for IPv. Web page.
Available at: http://www.faqs.org/rfcs/rfc2030.html
(cited on p. 46)

Minar, Nelson, Matthew Gray, Oliver Roup, Raffi Krikorian and Pattie Maes (2000). Hive: Distributed agents for networking things. *IEEE Concurrency* **8**(2): pp. 24-33. April-June 2000.
Available at: http://nelson.www.media.mit.edu/people/nelson/research/hive-asama99/hive-asama99.ps.gz
(cited on pp. 31, 68)

MindPixel (2000). MindPixel Digital Mind Modeling Project. Web page.
Available at: http://www.mindpixel.com/
(cited on p. 123)

Moran, Thomas P., Eric Saund, William van Melle, Anuj Gujar, Ken Fishkin and Beverly Harrison (1999). Design and technology for Collaborage: Collaborative collages of information on physical walls. In the *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST '99)*, pp. 197-206, Asheville, NC, ACM. November 7-10.
Available at: http://www.parc.xerox.com/spl/members/saund/papers/uist99.doc
(cited on p. 21)

Myers, Brad A. (1990). A new model for handling input. *Transactions on Information Systems* **8**(3): pp. 289-320. July 1990.
Available at: http://www.acm.org/pubs/articles/journals/tois/1990-8-3/p289-myers/p289-myers.pdf
(cited on p. 39)

Myers, Brad A. and David S. Kosbie (1996). Reusable hierarchical command objects. In the *Proceedings of the 1996 ACM Conference on Human Factors in Computing Systems (CHI '96)*, pp. 260-267, Vancouver, BC, ACM. April 13-18, 1996.
Available at: http://www1.acm.org/sigs/sigchi/chi96/proceedings/papers/Myers/bam_com.htm
(cited on p. 94)

Myers, Brad, Scott E. Hudson and Randy Pausch (2000). Past, present and future of user interface software tools. *ACM Transactions on Computer Human Interaction* **7**(1): pp. 3-28. March 2000.
Available at: http://www.cs.cmu.edu/~amulet/papers/futureofhci.pdf
(cited on pp. 9, 10)

Mynatt, Elizabeth D., Maribeth Back, Roy Want, Michael Baer and Jason B. Ellis (1998). Designing Audio Aura. In the *Proceedings of the CHI '98 Conference on Human Factors in Computing Systems*, pp. 566-573, Los Angeles, CA, ACM. April 18-23, 1998.
Available at: http://www.parc.xerox.com/csl/members/want/papers/aa-chi98.pdf
(cited on pp. 12, 17, 20)

mySQL (2000). mySQL. Web page.
Available at: http://www.mysql.com/
(cited on p. 53)

Nardi, Bonnie A., James R. Miller and David J. Wright (1998). Collaborative, programmable intelligent agents. *Communications of the ACM* **41**(3): pp. 96-104. March, 1998.
Available at: http://www.miramontes.com/ADD-CACM/ADD-CACM.html
(cited on p. 33)

Nelson, Giles J. (1998). Context-aware and location systems. PhD dissertation. University of Cambridge.
Available at: http://www.acm.org/sigmobile/MC2R/theses/nelson.ps.gz
(cited on pp. 19, 36, 122)

Newman, Neill J. (1999). Sulawesi: A wearable application integration framework. In the *Proceedings of the 3rd International Symposium on Wearable Computers (ISWC '99)*, pp. 170-171, San Francisco, CA, IEEE. October 20-21, 1999.
Available at: http://wearables.essex.ac.uk/reports/ISWC99/index.htm
(cited on p. 32)

Nguyen, David, Joe Tullio, Tom Drewes and Elizabeth Mynatt (2000). Dynamic Door Displays. Web page.
Available at: http://ebirah.cc.gatech.edu/~jtullio/doorshort.htm
(cited on p. 90)

Object Management Group (2000). OMG's CORBA Website. Web page.
Available at: http://www.corba.org/
(cited on pp. 36, 67)

Olsen, Dan R. (1992). *User interface management systems: Models and algorithms*. 1st edition. Morgan Kaufmann, San Mateo, CA.
(cited on pp. 39, 122)

Orr, Robert J. (2000). The Smart Floor: A mechanism for natural user identification and tracking. In the *Proceedings of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, pp. 275-276, The Hague, Netherlands, ACM. April 1-6, 2000.
Available at: http://www.cc.gatech.edu/fce/pubs/floor-short.pdf
(cited on pp. 8, 105)

Pandit, Milind S. and Sameer Kalbag (1997). The Selection Recognition Agent: Instant access to relevant information and operations. In the *Proceedings of the 1997 International Conference on Intelligent User Interfaces (IUI '97)*, pp. 47-52, Orlando, FL, ACM. January 6-9, 1997.
Available at: http://www.acm.org/pubs/articles/proceedings/uist/238218/p47-pandit/p47-pandit.pdf
(cited on p. 33)

Pascoe, Jason (1998). Adding generic contextual capabilities to wearable computers. In the *Proceedings of the 2nd IEEE International Symposium on Wearable Computers (ISWC'98)*, pp. 92-99, Pittsburgh, PA, IEEE. October 19-20, 1998.
Available at: http://www.cs.ukc.ac.uk/pubs/1998/676/content.zip
(cited on pp. 3, 5, 6, 8, 12, 19, 20, 37)

Pascoe, Jason, Nick S. Ryan and David R. Morse (1998). Human-Computer-Giraffe Interaction – HCI in the field. In the *Workshop on Human Computer Interaction with Mobile Devices*, Glasgow, Scotland. May 21-23, 1998.
Available at: http://www.dcs.gla.ac.uk/~johnson/papers/mobile/HCIMD1.html#_Toc420818982
(cited on pp. 5, 12)

Pederson, Elin R. and Tomas Sokoler (1997). AROMA: Abstract representation of presence supporting mutual awareness. In the *Proceedings of the 1997 ACM Conference on Human Factors in Computing Systems (CHI '97)*, pp. 51-58, Atlanta, GA, ACM. March 22-27, 1997.
Available at: http://www.acm.org/sigs/sigchi/chi97/proceedings/paper/erp.htm
(cited on pp. 12, 16)

Phillips, Brenton A. (1999). Metaglue: A programming language for multi-agent systems. MEng dissertation. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Cambridge, MA.
Available at: http://www.ai.mit.edu/projects/hal/brenton-thesis.pdf
(cited on p. 31)

Picard, Rosalind W. (1997). *Affective computing*. 2nd edition. MIT Press, Cambridge, MA.
(cited on p. 86)

PinPoint (1999). PinPoint 3D-iD introduction. Web page.
Available at: http://www.pinpointco.com/products/products_title.htm
(cited on p. 73)

Postel, Jonathan B. (1982). Simple Mail Transfer Protocol RFC 821. Web page.
Available at: http://info.internet.isi.edu/in-notes/rfc/files/rfc821.txt
(cited on p. 66)

Rabiner, Lawrence R. and Biing-Hwang Juang (1986). An introduction to hidden Markov models. *IEEE Acoustic, Speech, and Signal Processing Magazine* **3**(1): pp. 4-16. January 1986.
(cited on p. 122)

Rekimoto, Jun (1996). Tilting operations for small screen interfaces. In the *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology (UIST'96)*, pp. 167-168, Seattle, WA, ACM. November 6-8, 1996.
Available at: http://www.csl.sony.co.jp/person/rekimoto/papers/uist96.pdf
(cited on pp. 12, 13)

Rekimoto, Jun, Yuji Ayatsuka and Kazuteru Hayashi (1998). Augment-able reality: Situated communication through physical and digital spaces. In the *Proceedings of the 2nd IEEE International Symposium on Wearable Computers (ISWC'98)*, pp. 68-75, Pittsburgh, PA, IEEE. October 19-20, 1998.
Available at: http://www.csl.sony.co.jp/person/rekimoto/papers/iswc98.pdf
(cited on pp. 5, 12)

Rhodes, Bradley J. (1997). The wearable remembrance agent: A system for augmented memory. *Personal Technologies* **1**(4): pp. 218-224. December 1997.
Available at: http://rhodes.www.media.mit.edu/people/rhodes/Papers/wear-ra-personaltech/index.html
(cited on p. 93)

Rhodes, Bradley J. (2000). Just-in-time information retrieval. PhD dissertation. Media Lab, Massachusetts Institute of Technology. Cambridge, MA.
Available at: http://www.media.mit.edu/~rhodes/Papers/rhodes-phd-JITIR.pdf
(cited on p. 93)

Richardson, Tristan (1995). Teleporting - Mobile X sessions. In the *Proceedings of the 9th Annual X Technical Conference*, Boston, MA. January 1995.
Available at: ftp://ftp.uk.research.att.com/pub/docs/att/tr.95.7.html/paper.html
(cited on p. 20)

Rodden, Tom, Keith Cheverst, Nigel Davies and Alan Dix (1998). Exploiting context in HCI design for mobile systems. In the *Workshop on Human Computer Interaction with Mobile Devices*, Glasgow, Scotland. May 21-23, 1998.
Available at: http://www.dcs.gla.ac.uk/~johnson/papers/mobile/HCIMD1.html#_Toc420818967
(cited on pp. 3, 27)

Ryan, Nick (1997). MCFE metadata elements, version 0.2. Working document. University of Kent at Canterbury. Kent, UK.
Available at: http://www.cs.ukc.ac.uk/research/infosys/mobicomp/Fieldwork/Notes/mcfemeta.html
(cited on p. 5)

Ryan, Nick, Jason Pascoe and David Morse (1998). *Enhanced reality fieldwork: the context-aware archaeological assistant*. Computer Applications and Quantitative Methods in Archaeology. V. Gaffney, M. van Leusen and S. Exxon, Editors. Oxford.
Available at: http://www.cs.ukc.ac.uk/research/infosys/mobicomp/Fieldwork/Papers/CAA97/ERFldwk.html
(cited on pp. 3, 5, 12)

Salber, Daniel, Anind K. Dey and Gregory D. Abowd (1999a). The Context Toolkit: Aiding the development of context-enabled applications. In the *Proceedings of the 1999 ACM Conference on Human Factors in Computer Systems (CHI '99)*, pp. 434-441, Pittsburgh, PA, ACM. May 15-20, 1999.
Available at: http://www.cc.gatech.edu/fce/contexttoolkit/pubs/chi99.pdf
(cited on pp. 39, 44, 71, 74)

Salber, Daniel, Anind K. Dey, Robert J. Orr and Gregory D. Abowd (1999b). Designing for ubiquitous computing: A case study in context sensing. Technical Report GIT-GVU-99-29. Georgia Institute of Technology, GVU Center. Atlanta, GA.
Available at: ftp://ftp.cc.gatech.edu/pub/gvu/tr/1999/99-29.pdf
(cited on p. 5)

Salutation Consortium (2000). Salutation Consortium. Web page.
Available at: http://www.salutation.org
(cited on p. 67)

Sandhu, Ravi S., Edward J. Coyne, Hal L. Feinstein and Charles E. Youman (1996). Role based access control models. *IEEE Computer* **29**(2): pp. 38-47. February 1996.
Available at: http://www.list.gmu.edu/journals/computer/ps_ver/i94rbac.ps
(cited on p. 92)

Schilit, Bill N. (1995). System architecture for context-aware mobile computing. PhD dissertation. Columbia University. New York.
Available at: http://www.fxpal.xerox.com/people/schilit/schilit-thesis.pdf
(cited on pp. 28, 28, 35, 68)

Schilit, Bill N., Norman I. Adams and Roy Want (1994). Context-aware computing applications. In the *Proceedings of the 1st International Workshop on Mobile Computing Systems and Applications*, pp. 85-90, Santa Cruz, CA, IEEE. December 8-9, 1994.
Available at: ftp://ftp.parc.xerox.com/pub/schilit/wmc-94-schilit.ps
(cited on pp. 3, 5, 6, 20)

Schilit, Bill N. and Marvin M. Theimer (1994). Disseminating active map information to mobile hosts. *IEEE Network* **8**(5): pp. 22-32. September/October 1994.
Available at: ftp://ftp.parc.xerox.com/pub/schilit/AMS.ps.Z
(cited on pp. 3, 5, 123)

Schmidt, Albrecht, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven and Walter Van de Velde (1999). Advanced interaction in context. In the *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing (HUC '99)*, pp. 89-101, Karlsruhe, Germany, Springer-Verlag. September 27-29, 1999.
Available at: http://www.teco.edu/~albrecht/publication/huc99/advanced_interaction_context.pdf
(cited on p. 36)

Schmidt, Albrecht, Michael Beigl and Hans-Werner Gellersen (1998). There is more to context than location. In the *Interactive Applications of Mobile Computing (IMC '98)*, Rostock, Germany. November 24-25, 1998.
Available at: http://www.rostock.igd.fhg.de/~imc98/Proceedings/imc98-SessionMA1-1.pdf
(cited on p. 20)

Schwartz, Michael F., Alan Emtage, Brewster Kahle and B. Clifford Neuman (1992). A comparison of Internet resource discovery approaches. *Computer Systems* **5**(4): pp. 461-493. Fall 1992.
Available at: ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/RD.Comparison.ps.Z
(cited on p. 30)

Shardanand, Upendra and Pattie Maes (1995). Social information filtering: Algorithms for automating "word of mouth". In the *Proceedings of the 1995 ACM Conference on Human Factors in Computing Systems (CHI '95)*, pp. 210-217, Denver, CO. May 7-11, 1995.
Available at: http://www.acm.org/sigchi/chi95/Electronic/documnts/papers/us_bdy.htm
(cited on p. 83)

Signiform (2000). ThoughtTreasure home page. Web page.
Available at: http://www.signiform.com/tt/htm/tt.htm
(cited on p. 123)

Singh, Push (2000). OpenMind Commonsense. Web page.
Available at: http://openmind.media.mit.edu
(cited on p. 123)

Sun Microsystems (1999). Jini Connection Technology Home Page. Web page.
Available at: http://www.sun.com/jini/
(cited on pp. 57, 67)

Sun Microsystems (2000a). Java Remote Method Invocation (RMI). Web page.
Available at: http://www.javasoft.com/j2se/1.3/docs/guide/rmi/index.html
(cited on pp. 31, 67)

Sun Microsystems (2000b). JavaSpaces Technology. Web page.
Available at: http://www.javasoft.com/products/javaspaces/index.html
(cited on p. 68)

Sun Microsystems (2000c). JDBC Data Access API. Web page.
Available at: http://www.javasoft.com/products/jdbc/
(cited on p. 53)

SVRLOC Working Group of the IETF (1999). Service Location Protocol Home Page. Web page.
Available at: http://www.srvloc.org/
(cited on p. 67)

Universal Plug and Play Forum (2000). Universal Plug and Play Forum Home Page. Web page.
Available at: http://www.upnp.org
(cited on p. 67)

Want, Roy, Andy Hopper, Veronica Falcao and Jonathan Gibbons (1992). The Active Badge location system. *ACM Transactions on Information Systems* **10**(1): pp. 91-102. January 1992.
Available at: http://www.parc.xerox.com/csl/members/want/papers/ab-tois-jan92.pdf
(cited on pp. 5, 8, 12, 15, 29)

Want, Roy, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis and Mark Weiser (1995). The PARCTAB ubiquitous computing experiment. Technical Report CSL-95-1. XEROX Palo Alto Research Center. Palo Atlo, CA.
Available at: http://sandbox.xerox.com/parctab/csl9501/paper.html
(cited on pp. 20, 88)

Ward, Andrew M. R. (1998). Sensor-driven computing. PhD dissertation. Computer Laboratory, University of Cambridge. Cambridge, UK.
Available at: http://www.acm.org/sigmobile/MC2R/theses/ward.pdf
(cited on p. 20)

Ward, Andy, Alan Jones and Andy Hopper (1997). A new location technique for the active office. *IEEE Personal Communications* **4**(5): pp. 42-47. October 1997.
Available at: http://www.it.kth.se/edu/Ph.D/LocationAware/ftp.orl.co.uk:/pub/docs/ORL/tr.97.10.pdf
(cited on pp. 3, 5)

Weiser, Mark (1991). The computer for the 21st Century. *Scientific American* **265**(3): pp. 66-75. September 1991.
Available at: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html
(cited on p. 1)

Weiser, Mark and John Seely Brown (1997). *The coming age of calm technology*. Beyond Calculation: The Next Fifty Years of Computing. pp. 75-85. Peter J. Denning and Robert M. Metcalfe, Editors. Springer-Verlag, New York.
Available at: http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm
(cited on p. 2)

Whitehead, E. James, Rohit Khare, Richard N. Taylor, David S. Rosenblum and Michael M. Gorlick (1999). Architectures, protocols, and trust for info-immersed active networks. In the *DARPA/NIST/NSF Workshop on Research Issues in Smart Computing Environments*, Atlanta, GA. July 25-26, 1999.
Available at: http://www.dyncorp-is.com/darpa/meetings/sce99jul/papers/RosenblumD.pdf
(cited on p. 23)

Winograd, Terry (2001). *Interaction spaces for 21st century computing*. Human-Computer Interaction in the New Millenium (in press). John Carroll, Editor. Addison-Wesley, Reading, MA.
(cited on pp. 37, 68)

Yan, Hao and Ted Selker (2000). Context-aware office assistant. In the *Proceedings of the ACM International Conference on Intelligent User Interfaces (IUI 2000)*, pp. 276-279, New Orleans, LA, ACM. January 9-12, 2000.
Available at: http://www.media.mit.edu/~lieber/IUI/Yan/Yan.pdf
(cited on p. 90)

Yang, Jie, Weiyi Yang, Matthias Denecke and Alex Waibel (1999). Smart Sight: A Tourist Assistant System. In the *Proceedings of the 3rd IEEE International Symposium on Wearable Computers (ISWC '99)*, pp. 73-78, San Francisco, CA, IEEE. October 18-19, 1999.
Available at: http://www.is.cs.cmu.edu/papers/multimodal/ISWC99/ISWC99-jie.pdf
(cited on p. 14)

VITA

Anind Kumar Dey was born on September 29, 1970 in Salmon Arm, British Columbia, Canada. He attended high school at Salmon Arm High School in Salmon Arm, British Columbia, where he grew up. He received his Bachelor of Applied Science degree in Computing Engineering at Simon Fraser University in Burnaby, British Columbia in 1993. He received his Masters of Science degree in Aerospace Engineering at the Georgia Institute of Technology in 1995.

During his graduate work, Anind has served as a Research Assistant in the College of Computing, working on a number of projects related to context-aware computing. He has also worked as a teaching assistant and consultant for a number of companies including Motorola and MCI. Anind has also done internships at Motorola (Schaumburg, Illinois) and Interval Research (Palo Alto, California). During his time at the Georgia Institute of Technology, he received Motorola's University Partnership in Research award/funding (1996-2000) and was named the Outstanding Graduate Research Assistant (2000).