Providing Architectural Support for Building Context-Aware Applications

A Dissertation Proposal
Presented To
The Faculty of the Division of Graduate Studies

By

Anind K. Dey

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

College of Computing
Georgia Institute of Technology
August 1999

Contents

ABSTRACT

Computers generally do not have access to situational information or context. Context can from users and from the environment. By providing access to context, we can improve the ability of computers to react to changes in the environment and to support us in changing situations. In this thesis proposal we provide further motivation for the use of context. We provide a definition of context and context-awareness and discuss some important features of both. We show that because context-aware applications are usually built in an *ad hoc* manner, they are difficult to build and evolve. We identify a novel and simple design process for building context-aware applications and discuss the necessary features of an architecture that supports the building process. We present 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.*

We describe the work that we have completed in proving this thesis statement and the work remaining. We present the goals of our research and present our expected contributions in the understanding of context, supporting the building and evolving of context-aware applications and providing a framework for investigating complex research issues in context-aware computing. Finally, we provide a timeline for completing this thesis research.

1 INTRODUCTION: Motivation for Context-Aware Computing

Humans are quite successful in conveying ideas to each other and reacting appropriately. This is due to many factors: e.g. 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 implicit situational information, or *context*, to increase the conversational bandwidth. Unfortunately this ability to convey ideas does not transfer well to humans interacting with computers. Computers do not understand our language, do not understand how the world works and have no implicit understanding of situations. In traditional interactive computing, users have an impoverished mechanism for providing input to computers, using a keyboard and mouse. 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 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 them 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. Then 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 fields are attempting to address this input deficiency but they step mainly from 2 basic directions:

- improving the language that humans can use to interact with computers,
- 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 where 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, relationships to these other people, etc., is crucial to understanding what is occurring. The process of building this shared understanding is called grounding [Clark91]. Since both 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 is making it easier to input explicit information and the second technique is allowing the use of unused implicit information that can be vital to understanding the explicit information. It is this second technique that we are primarily interested in. We are attempting to use context as an implicit cue to enrich the impoverished interaction from humans to computers.

So 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 everything relevant to a given situation. However, the goal of *context-aware computing*, applications that use context, should be to make interacting with computers easier. Forcing users consciously to increase the amount of information they have to input is making 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, not harder. Weiser coined the term "ubiquitous computing" to describe computing that is invisible in use [Weiser91], and 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 [Weiser96]. Here, the idea is to make interacting with computers and the environment easier, allowing users to not have to think consciously about using the computers. 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. Mobile computing and ubiquitous computing have given users the expectation that they can access information and services whenever 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. 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 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.

Applications that use context, whether on a desktop or in a mobile or ubiquitous computing environment, are called context-aware. These types of 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.

2 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 effectively use context, 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. We have completed an extensive survey on the field of context-aware computing. From this survey, we produced new definitions for context and context-awareness [Dey99b]. We also attempted to create categories of context and context-aware features. In this section, we will present the definitions and important features.

2.1 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 user is doing) 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. 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').

There are certain types of context that are, in practice, more important than others. These are ***location***, ***identity***, ***activity*** and ***time***. Location, identity, time, and activity are the *primary* 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, birthdate, 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. From these examples, it should be evident that the primary pieces of context for one entity can be used as indices to find *secondary* context (e.g., the email address) for that same entity as well as primary context for other related entities (e.g., other people in the same location).

In this initial categorization, we have a simple two-tiered system. The four primary pieces of context already identified comprise the first level. All the other types of context are on the second level. The secondary pieces of context share a common characteristic: they can be indexed by primary context because they are attributes of the entity with primary context. For example, a user's phone number is a piece of secondary context and it can be obtained by using the user's identity (primary context) as an index into an information space like a phone directory. There are some situations in which multiple pieces of primary context are required to index into an information space. For example, the forecasted weather is context in an outdoor tour guide that uses the information to schedule a tour for users. To obtain the forecasted weather, both the location (primary context) for the forecast and the date of the desired forecast (primary context) are required.

This first attempt at a categorization of context is clearly incomplete. It does not include hierarchical or containment information and we have found examples where the primary-secondary distinction is quite blurred. An example of hierarchical or containment information 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 [Schilit94b]. It is not clear how our categorization helps to support this notion of hierarchical knowledge. An example of primary-secondary blurring is with a person's identity. Is a person identified by their name, a phone number, or an email address? Which of these is primary context and which is secondary context. Each of these identity pieces can be used to derive the other. We will continue our effort to attain a better understanding of context and how it is used and to identify the orthogonal dimensions of context.

2.2 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.*

The context-aware features we have identified are

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.

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. An example of the second feature is when the user prints a document and it is printed on the closest printer to the user. 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.

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.

3 MOTIVATION FOR OUR RESEARCH: Why Context-Aware Applications are Difficult to Build

In the first section, we described why context-aware computing is an interesting and relevant field of research in computer science. In the second section, we identified definitions and features of context and context-awareness to provide a better understanding of the scope of the applications that we would like to support. In this section, we will discuss why these applications have traditionally been so difficult to build.

What has hindered applications from making greater use of context and from being context-aware? 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 [Nelson98]. This results in a lack of generality, requiring each new application to be built from the ground up. To understand this difficulty, we need to examine the design process for building these applications.

3.1 Design Process

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 [Abowd99]) is as follows:

1 *Specification*: Specify the problem being addressed and a high-level solution
- 1a Specify the context-aware behaviors to implement
- 1b Determine what context is required for these behaviors

2 *Acquisition*: Determine what hardware or sensors are available to provide that context.
- 2a Install the sensor on the platform it requires.
- 2b Understand exactly what kind of data the sensor provides.
- 2c If no application programming interface (API) is available, write software that speaks the protocol used by the sensor.
- 2d If there is an API, learn to use the API to communicate with the sensor.
- 2e Determine how to query the sensor and how to be notified when changes occur.
- 2f Interpret the context, if applicable.

3 *Delivery*: Provide methods to support the delivery of context to one or more, possibly remote, applications.

4 *Reception*: Work with the context.
- 4a Receive or request the context.
- 4b Convert it to a useable form through interpretation
- 4c Analyze the information to determine usefulness.

5 *Action*: If context is useful, perform context-aware behavior.
- 5a Analyze the context treating it as an independent variable or by combining it with other information collected in the past or present.
- 5b Choose context-aware behavior to perform

We will show that for these first four steps, there are general infrastructure-level supporting mechanisms that are required by all but the most trivial context-aware applications. The last step is application/service-specific and we may not be able to identify or apply general mechanisms to support them. It is important to note that the goal of a context-aware application designer is to provide context-aware services. The designer does not want to worry about the initial steps, but instead would like to concentrate on the actual context-aware behaviors. Furthermore, it is these initial steps that make building context-aware applications difficult and time-consuming.

3.1.1 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 1a). Also, the developer determines what context is needed. In this case, the relevant context is location, identity and time (step 1b). The second step, *acquisition*, is where the developer deals directly with the sensors. Sensors can be hardware or software-based, providing both real-world context such as location and identity, and virtual context such as the name of the last file a user read. Java iButtons™ [DS98] 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 PC, at the entrance to the research lab (step 2a). 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 2b). The location context is simply the location of the installed reader, and time context is determined from the PC 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 2b/c). This code must support both querying and notification of events (step 2e), where, informally, an event is a timely difference that makes a difference [Whitehead99]. 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 final step in acquisition is to interpret the context, if possible and applicable. 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 connected to 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, 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 queries the sensor. It wants to maintain accurate state information, so it also subscribes to the iButton™ asking to be notified of relevant changes (step 4a). 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 4b). 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 4c). 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 5a). Finally, the developer can act on the context, and display the user's new state on the In Out Board (step 5b).

3.2 Essential and Accidental Activities

In his famous essay on software engineering practices, Fred Brooks distinguished essential and accidental activities in software development [Brooks87]. Essential activities are fundamental to constructing a piece of software and include understanding a problem and modeling a solution for that problem. Accidental tasks are ones that are only necessary because support is not available to handle or take care of them. 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.

3.2.1 Specification

In step one, 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 Acquisition

In step two, 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 global positioning system (GPS) receiver and a smart card with accompanying reader may both appear to provide location context. In fact, the GPS receiver does provide 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.

3.2.3 Delivery

Once in the correct format, the context information must be delivered to the application. 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.

3.2.4 Reception

In the reception step, 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 its usefulness. 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.

3.2.5 Action

If the context is useful information, the application designer must provide functionality to further analyze the context to determine what action to take. This ranges from analyzing the types of context received to analyzing the actual values of the context. The action includes choosing a context-aware behavior to perform as well as indicating how the behavior should be executed. This action step is application-specific and is essential. It is or should be the main emphasis in building context-aware applications, actually acting on the context.

3.3 Resulting Problems

Our previous assessment of context-aware software application building as "*ad hoc*" may appear to be contradictory to the existence of a design process. It is, in fact, the implementation of the accidental steps of the design process that is *ad hoc*, due to the difficulties we expressed in each of the above steps. Pascoe wrote that it is a hard and time-consuming task to create software that can work with variety of software to capture context, translate it to meaningful format, manipulate and compare it usefully, and present to user in meaningful way [Pascoe98]. 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. As a result of this *ad hoc* implementation, a general trend of tightly connecting applications and sensors has emerged that operates against the progression of context-aware computing as a research field. This trend has led to four general problems:

- 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.

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

3.3.1 General Lack of Context-Aware Applications

Due to the manner in which context-aware applications are built, designers put little effort into making their sensors reusable by other designers and applications. This has resulted in the lack of basic context-aware applications. We believe that the ability to leverage off of the sensor work of others is fundamental to building context-aware applications. Because many sensor solutions were not designed to be reused, they are very difficult to integrate into existing applications that do not already use context, as well as being difficult to add to existing context-aware applications.

3.3.2 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. The reason for this 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 shown by the number and difficulty of the sub-steps in Step 2 of the design process given above.

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 are a burden to the application programmer, and 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 done 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 [Want92], but built their own badge system with greater functionality – the ParcTab [Want95]. In all of

their context-aware work, they used the ParcTab as the main source of user identity and location [Schilit94a, Schilit95, Mynatt98]. The wearable computing group at Oregon always uses an infrared positioning system to determine location [Bauer98, Korteum98]. 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 [Richardson95, Harter94, Harter99, Adly97]. 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, to create 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 [Pascoe98]. 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) [Ward98]. However, this doesn't seem fruitful – the diversity of context that application designers want to use can not be captured with a handful of sensors. Instead, we should provide support to allow application designers to make use of new sensors.

3.3.3 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 [Schmidt98, Dey99b]. 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.

3.3.4 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, this is 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 take the previously described hypothetical In Out Board (with no infrastructure support) as an example. If the reader that acquires identity information were placed at the entrance to the building, as opposed to the entrance to the laboratory, the application would have to be stopped, its code modified to use the sensor at its new location and then restarted. This is a minor difficulty, 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. This also applies when the designer changes or adds to the context being used. We will revisit the In Out Board application again. If a face recognition system were used to acquire identity rather than an iButton™ and reader, 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 face recognition system. The portion of the application that performs conversions on the sensed context and determines usefulness will also require rewriting. The difficulties in adapting applications to context acquisition changes results in relatively static applications. This leads to applications which are short-lived in duration, which is opposed to the view in ubiquitous computing where computing services are available all the time (long-term consecutive use). The inability to easily evolve applications does not aid the progress of context-aware computing as a research field.

3.4 Overview of the Difficulty in Building Context-Aware Applications

We have described the design process for context-aware computing, concentrating on the specification, acquisition and delivery of context. We have shown that when these steps 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
- 1a Specify the context-aware behaviors to implement
- 1b Determine what context is required for these behaviors

2 *Acquisition*: Determine what hardware or sensors are available to provide that context and install them.

3 *Action*: If context is useful, perform context-aware behavior.
- 3a Analyze the context treating it as an independent variable or by combining it with other information collected in the past or present.
- 3b Choose context-aware behavior to perform

To help designers migrate to this reduced design process, 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 four basic problems that have emerged due to this trend: the general lack of context-aware applications, 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.*

4 RELATED WORK

We will now describe previous attempts to provide support for both the original design process and the simpler design process. We will discuss how each attempt has failed to support the design process and failed to alleviate one or more of the four basic problems identified in the previous section.

4.1 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. The second problem with this technique is that it does not support good software engineering practices. The technique does not enforce separation of concerns between application semantics and the low-level details of context acquisition from individual sensors. This leads to a loss of generality, making the sensors difficult to reuse in other applications and difficult to use simultaneously in multiple applications.

4.1.1 Manipulative User Interfaces

In the manipulative user interfaces work [Harrison98], 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.

4.1.2 Tilting Interfaces

In the similar tilting interfaces work [Rekimoto96], 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.

4.1.3 Cyberguide

The Cyberguide system provided a context-aware tour guide to visitors to a "Demo Day" at a research laboratory [Abowd97a, Long96]. The tour guide is the most commonly developed context-aware application. 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. 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. 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.

4.2 Use of Sensor Abstractions

In this section, we will discuss systems that have used a sensor abstraction 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. They provide no support for notification nor any support or guidelines for context acquisition. 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 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. This results in limited use of sensors, context types, and the inability to evolve applications.

4.2.1 Active Badge

The original Active Badge call-forwarding application is perhaps the first application to be described as being context-aware. In this application, users wore Active Badges [Want92], infrared transmitters that transmitted an 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 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.

4.2.2 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 [Cooperstock97]. 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 determine 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 "awareness" of activity around a specific device. 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. This requires the application to deal with each daemon in a distinct fashion, affecting both evolution and the use of new sensors and context types.

4.3 Beyond Sensor Abstractions

In this section, we discuss systems that not only support sensor abstractions, but also support additional mechanisms such as notification, storage, or interpretation. The problems with these systems is that they do not provide support for the entire design process.

4.3.1 AROMA

The AROMA project attempted to provide peripheral awareness of remote colleagues through the use of abstract information [Pederson97]. Its object-oriented architecture used the concepts of sensor abstraction and interpretation. It had a minimal notion of storage, keeping only the last n values in a circular buffer. 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.

4.3.2 metaDesk

The metaDesk system was a platform for demonstrating tangible user interfaces [Ullmer97]. Instead of using graphical user interface widgets to interact with the system, users used physical icons to manipulate information in the virtual world. The system used sensor proxies to separate the details of individual sensors from the application. This system architecture supported distribution and a namespaces mechanism to allow simple runtime evolution of applications. Only a polling mechanism was provided and interpretation was left to individual applications.

4.3.3 Limbo

Limbo is an agent-based system that uses quality of service information to manage communication channels between mobile fieldworkers [Davies97]. 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 [Friday96]. Other agents place (and remove) service-related information into (and from) the tuple spaces. These spaces provide an abstraction of the sensor details, only providing access to the sensor data. This technique supports distributed sensing, limited interpretation and limited storage.

4.3.4 NETMAN

The NETMAN system is a collaborative wearable application that supports the maintenance of computer networks in the field [Korteum98]. It uses location, object identities and network traffic context to provide relevant information to the field worker and the assisting remote network expert. The system uses sensor proxies to abstract the details of the sensors from the application and also handles the delivery of context through a subscription-based mechanism. There is no support for acquiring sensor information, making it difficult to add new sensors. As well, interpretation is left to the application using the context.

4.3.5 Open Agent Architecture

The Open Agent Architecture is an agent-based system that supports task coordination and execution [Cohen94]. 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 consecutive use nor was it meant to handle a large number of applications, services, and sensors.

4.3.6 Audio Aura

In the Audio Aura system, location and identity context was used to provide awareness about physical and virtual information, using serendipitous background audio [Mynatt98]. For example, when a user enters a social area, they receive an audio cue 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.

4.4 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 and problems, but, in fact, deal with only a portion of the context-aware application problem space.

4.4.1 Stick-e Notes

The Stick-e notes system is a general framework for supporting a certain class of context-aware applications [Brown96]. 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 to be triggered when the right

combination of context is achieved. 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", follows:

<note>
    <required>
    <at> (1,4) .. (3,5)
    <facing> 150 .. 210
    <during> December
<body>
    The large floodlit building at the bottom of the hill is cathedral. [Brown97]

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 limited to simple boolean ANDs and can not 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. While no information is provided on how this mechanism acquires its context, the mechanism does hide the details of the acquisition from applications.

4.4.2 CyberDesk

In our previous research on context-aware computing, we built an architecture called CyberDesk [Dey97, Abowd97b, Dey98a, Dey99a]. This architecture was built to automatically integrate web-based services based on virtual context. The virtual context was the personal information the user was interacting with on-screen including email addresses, mailing addresses, dates, names, etc. An example application is when a user is looking at her schedule for the day and sees that she has a meeting in the afternoon with an acquaintance. She highlights that person'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, looking up a relevant phone number using the web, and sending email to the acquaintance.

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. The architecture provided support for the entire simpler design process identified in Section three. Applications simply specified what context types they were interested in, and were notified when those context types were available. The modular architecture handled automatic interpretation, 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 [Dey98b].

4.4.3 Schilit's System Architecture

In his Ph.D. thesis, Schilit presented a system architecture that supported context-aware mobile computing [Schilit95]. 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 focused on making context-aware computing applications possible to build. From our survey of context-aware computing, we have seen that designers are indeed now capable of building context-aware applications, thanks in a large part to Schilit's work. 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. 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.

4.4.4 CALAIS

CALAIS, the Context And Location Aware Information Service, was another architecture that was designed to support context-aware applications [Nelson98]. This work was performed to solve two problems: the *ad hoc* nature of sensor use and the lack of a fine-grained location information management system. An abstraction was developed to hide the details of sensors from context-aware applications, but 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 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.

4.5 Proposed Systems

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

4.5.1 Situated Computing Service

The proposed Situated Computing Service has an architecture that is similar to CyberDesk for supporting context-aware applications [Hull97]. 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.

4.5.2 Context Information Service

The Context Information Service (CIS) is another proposed architecture for supporting context-aware applications [Pascoe98]. 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.

4.6 Overview of Related Work

In this section, we have presented previous 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 hard 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. 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. Finally, we presented architectures that were designed to support the building process. These architectures are limited to dealing with a portion of the process or are merely speculative.

5 ARCHITECTURE FEATURES

In Section four, we presented previous work in the area of architectural support for context-aware computing. While no one system provided a complete solution for supporting our reduced design process, each offered 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. In this section, we will present the features for an architecture that will both support the design process and provide runtime support for multiple simultaneously executing context-aware applications. The identification of these features stem from our own work in context-aware computing as well as from a survey we have performed on the field. We will present the features in two parts. First, we will present the set of minimal features required by an architecture for supporting the design and execution of context-aware applications. Then, we will present architectural features that, while not required, are useful for supporting the reduced design process and reducing the tight connectivity typically found between applications and sensors.

5.1 Requirements

Following is the minimal set of requirements for an architecture to support the design and execution of context-aware applications. All but the most trivial context-aware applications will require the following features:

- support for distribution of context and sensing;
- support for event management;
- support independence and availability of components that collect context; and,
- support for the interpretation of context.

We see many of these features in the systems presented in the related work. We will now discuss these features.

5.1.1 Support for the Distribution of Context and Sensing

Traditional user input comes from the keyboard and mouse. These devices are connected directly to the computer they are being used with. 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. Support for the distribution of sensing and context is our first high-level requirement.

5.1.2 Support Event Management

In our review of related work, we discussed systems 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 [Rodden98]. Querying a sensor for context is appropriate for one-time context needs. But the sole use of querying requires that applications be proactive 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.

5.1.3 Support Independence and Availability of Components that Collect Context

With GUI applications, user interface components such as buttons and menus are instantiated, controlled and used by only a single application (with the exception of some groupware applications). In contrast, context-aware applications should not instantiate individual components that provide sensor data, but must be able to access existing ones, when they require. 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. Because they 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. Take the call-forwarding example from the Active Badge research [Want92]. When a phone call was received, an application tried to forward the call to the phone nearest the intended recipient. The application could not locate the user if the Badge server was not active. If the Badge server were instantiated and controlled by a single application, other applications could not use the context it provides.

5.1.4 Support for the Interpretation of Context

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 goes through before it reaches 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 is interpreted to determine where various users are and identity information is used to check co-location. At the next level, this information is 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 must 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 architecture. 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. For example, as discussed in the related work section, the CyberDesk infrastructure converts selected text to a name and then to an email address.

5.2 Useful Features

Following is a set of additional architectural features that, while not required, simplify the design of context-aware applications. They are:

- support the storing of context history;
- support for the aggregation of context information;
- support for transparent communications;
- support for resource discovery;
- support for using and building of abstractions;
- support for flexibility; and,
- use of language and platform-independent mechanisms.

We will now discuss each in turn.

5.2.1 Support for the Storing of Context History

A useful feature of context acquisition components linked to the need for constant availability is the desire to maintain historical information. User input widgets maintain little, if any, historical information. For example, a file selection dialog box keeps track of only the most recent files that have been selected and allows a user to select those easily. In general though, if a more complete history is required, it is left up to the application to implement it. In comparison, a context widget [Salber99], a component that collects context information, should maintain a history of all the context it obtains. A context widget may collect context when no applications are interested in that particular context information. Therefore, there are 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 this reason, context widgets should store their context.

5.2.2 Support for the Aggregation of Context Information

To facilitate the building of context-aware applications, our architecture should support the aggregation of context about entities in the environment. It is often the case that an application requires multiple pieces of information about a single entity. With the architecture described so far, an application would have to communicate with several different sensors to collect the necessary context about an interesting entities. This adds complexity to the design and negatively impacts maintainability. 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 has to use a combination of subscriptions and queries on different sensors to determine when the conditions are met. This is unnecessarily complex and is difficult to modify if changes are required. An architectural component that supports aggregation is responsible for collecting all the context about a given entity. With aggregation, 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 readers installed in multiple buildings. The In Out Board for a particular building is not interested where within a building a dock occurred, 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.

5.2.3 Support for Transparent Communications

In the previous sub-section on requirements, we presented a requirement to handle distributed context information. A useful feature is to make the communications between distributed sensors and applications transparent to both parties. 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.

5.2.4 Support for Resource Discovery

In order for an application to communicate with a sensor (or its proxy), it must know where the sensor is located and how to communicate with it (protocol and mechanisms to use). For distributed sensors, this means knowing both the hostname and port of the computer the sensor is running on. To be able to effectively hide these details from the application, the architecture needs to support a form of resource discovery [Schwartz92]. 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 docks inside the building.

An additional useful feature would be to extend this ability to aggregation components to help automatically find context relevant to the particular entity involved, and for use with interpreters to allow automatic interpretation of context. For the aggregator, this feature would save the application designer from having to explicitly state what sensor information was relevant. For example, a component that is responsible for aggregating all the context about a particular building X could specify to the resource discovery mechanism that it was interested in any information about ‘building X’ and it would simply receive it. For components that need interpreted information, this feature would automatically provide any

needed interpretation, saving the designer from having to explicitly request it. For example, in the In Out Board application, the application is interested in knowing in and out status of users, not that they docked in the building. With resource discovery, the application could indicate that it was interested in in/out status, and have the interpretation from user docking to this status automatically performed. This discussion assumes, of course, that the interpretation is available from the architecture.

5.2.5 Support for the Using and Building of Abstractions

So far, we have identified three important conceptual building blocks: acquisition of context from sensors, interpretation and aggregation. The architecture should contain components that implement each one of these building blocks or abstractions. To make it easy for applications to take advantage of these abstractions, each abstraction should support a standard interface. This will allow applications (and other components) to communicate with all aggregators the same way, all interpreters the same way, and all context acquirers the same way.

But there is a second important part to this useful feature and that is providing support for the building of components that implement these abstractions. In most of our previous discussions, we have assumed that the necessary components were available. If we needed a component that provides identity, or an interpreter that converts docking events to in/out status, or an aggregator that collects all the location information for a building, we assumed that such a component already existed and we used it. But what happens if such a component does not exist? Then, it is left up to the application designer 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 implementation within the application. By providing standard interfaces for each of the building blocks, we can provide this support.

For example, we have described required and useful features that all components should have. These include transparent communications for distributed components, resource discovery, availability and independent execution. In addition, for the acquisition of context, the necessary features are context storage and support for event management. For the interpretation abstraction, the obvious feature is support for interpretation. For the aggregation abstraction, the necessary features include those for the abstraction of sensor data into context as well as aggregation. What this means is that when a designer wants to build one of these components, the relevant features are implemented and automatically available for the designer to use. This leaves the designer with the much simpler task of designing only the task-specific pieces of the component. For example, to build an aggregation component that represents all the context about an building, the designer should only have to provide the building's identity. All other details should be taken care of for the designer.

5.2.6 Support for Flexibility

We need to provide 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 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. Each of the required and useful features should have easily replaceable/pluggable implementations.

5.2.7 Use of Language and Platform-Independent Mechanisms

Another feature that is useful for integrating the context architecture into existing applications is the use of programming language and computer platform-independent mechanisms. By having few requirements on the support the architecture requires 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.

5.3 Revisiting the Design Process

In Section three, we presented a design process for building context-aware applications and showed how we could reduce the design process by removing accidental activities. In this section, we have been presenting the requirements and useful features for an architecture to support the building of context-aware applications. We will now describe how an architecture that supports these features will allow a designer to use the simpler design process.

5.3.1 Specification

Specification of what context an application requires is an essential step and can not be reduced. However, via resource discovery, an application designer can quickly determine what context is available for use in the chosen environment.

5.3.2 Acquisition

If a sensor has not been used before, the acquisition of context is a necessary step. Installation of the sensor is supported by allowing the sensor to be installed on almost any platform and used with a number of programming languages. The writing of the code to actually acquire context is simplified by the provision of a general interface to implement. The interpretation abstraction allows the conversion of the sensor data into a more usable form. The extensions with resource discovery, described earlier, allow the interpretation to occur automatically with no work required on the part of the application designer. 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, 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 on the designer's part.

5.3.3 Delivery

The entire activity of delivering context is accidental. Many features of the architecture work in concert to allow this step to be removed. 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 specified by the application. Support for distribution and transparent communications mechanisms allow context to be sent between sensors and applications without the designer having to worry about either side.

5.3.4 Reception

Reception of context is also an accidental step. The architecture-level features can be used to remove this step. Resource discovery allows applications to automatically access 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. As described in the delivery sub-section, application designers do not have to deal with distribution or communication details to obtain context. Querying and notification mechanisms allow applications to retrieve the context they require under the conditions they specify. If the notification/subscription mechanisms are powerful enough, there is no need to analyze the information to determine usefulness. Conditions set when creating a subscription should contain all the information necessary to determine usefulness.

5.3.5 Action

Action is an application-specific and essential step. The features discussed in this section provide no assistance with this step.

5.4 Revisiting Problems Due to *Ad Hoc* Development

In Section three, we presented four problems that resulted from the use of *ad hoc* techniques in building context-aware applications. We will show how the architecture features presented in this section can be used to address these problems.

5.4.1 Lack of Context-Aware Applications

The two reasons given for the lack of context-aware applications were the inability to reuse sensors and the difficulty in integrating sensors into applications. Providing guidance to application designers on how to build sensor components supports reuse. The use of simple abstractions with common interfaces and support for the details that designers do not want to deal with (communications, distribution, storage, etc.) helps with this guidance. The automatic handling of these features also aids in integrating the architecture with existing applications. As well, the fact that applications do not have to worry about instantiating sensors, due to their independent and persistent execution, eases integration difficulties. Furthermore, by letting application designers replace default implementations of features with their own implementations allows the designers to leverage off of their own knowledge and experiences. Finally, the interoperability of components and applications written in different programming languages and on different platforms makes it much easier for applications to leverage off of existing infrastructure.

5.4.2 Lack of Variety of Sensors

While we want to support reuse of sensors, we do not want reuse to exist at the exclusion of other sensors and sensor types. By handling many of the "low-level" details involved in sensor development, such as communications, storage, notification and querying mechanisms, and distribution, we can simplify the design process for adding and using new sensors.

5.4.3 Lack of Variety of Context Types

By making it easier to use new sensors, we are making it easier to use new types of context in applications. In addition, the ability to aggregate and interpret context allows us to derive higher levels of context information from the low-level context usually acquired directly from sensors. The use of a wider variety of context types leads to more complex and more interesting context-aware applications.

5.4.4 Inability to Evolve Applications

The three difficulties in evolving context-aware applications are dealing with moving sensors, changing sensors and changing context. By using components that execute independently of the application, we can easily move or reconfigure these components without changing the application. The use of resource discovery keeps the movement transparent to the application, requiring no changes. When sensors are removed that do not impact the application, the same argument applies. If for some reason, a sensor that provides an important piece of context is removed, the architecture will use resource discovery to try and find another sensor that can provide the same information. If one can not be found, the application is notified. When sensors are added that provide context the application has asked for, this context information is automatically provided to the application. This ability is supported through a combination of the event management mechanisms and resource discovery. Finally, if a designer wants to use additional context in an application, he simply needs to specify what context he requires and under what conditions. The architecture takes care of the details.

5.5 Overview of Architecture Features

In this section, we have presented a set of features that are necessary in any architecture that supports the building of context-aware applications. We presented reasons why these features were required. These features are:

- support for distribution of context;
- support event management mechanisms;
- support independence and availability of components that collect context; and,
- support for the interpretation of context.

To this list, we presented a set of features that would be useful in such an architecture, to make the design or applications easier. These useful features are:

- support the storing of context history;
- support for the aggregation of context information;
- support for transparent communications;
- support for resource discovery;
- support for using and building abstractions (context acquirers, interpreters, and aggregators);

- support for flexibility; and,
- use of language and platform-independent mechanisms.

We discussed how these two sets of features, required and useful, can be used to reduce the design process to a set of essential steps and how they help in allowing designers to more easily build and evolve more complex context-aware applications.

6 Research Goals

In this thesis proposal, we have demonstrated the value of context in interactive computing. We presented definitions of context and context-awareness and showed how they impact our research in context-aware computing. Next, we described why building context-aware applications is a difficult process and identified a new simpler design process for building these applications. We reviewed previous context-aware applications and architectures and described their shortcomings in terms of this new design process. Next, we identified a set of required and useful features which an architecture that supports context-aware applications should support. We then described how these features support the use of the new design process. In this section, we will analyze our thesis statement that has been the focus of our research and detail our research goals and expected contributions.

6.1 Thesis Statement

Our thesis statement and the focus of our research in context-aware computing 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.*

We will now decompose this general thesis statement into its underlying hypotheses. Through a detailed study of context-aware computing and from our experience in building context-aware applications, we will be able to identify useful abstractions for dealing with context. In doing so, we will also gain a better understanding of what context is important and how we can represent it. Through the implementation of these abstractions and some underlying support, we will have a framework that helps application designers to build context-aware applications. In particular, the framework will enable designers to both easily build and evolve 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. A final hypothesis is that the framework will contain lightweight integration mechanisms and will be flexible enough to allow application designers to readily use it.

6.2 Research Goals

Our research has three main pieces. The first is an in-depth investigation of context and context-aware applications to enable us to identify the necessary mechanisms required for context-aware computing and to define a taxonomy of context. The second part is the actual implementation of the identified mechanisms in an architecture. This includes building to support flexibility and lightweight integration. The third part is to show that the architecture is extensible and can be used as an open research platform. This includes using it to experiment with new mechanisms and implementations for dealing with particular problems (privacy, security, uncertainty in data and group context, for example) in context-aware computing and to build new types of context-aware applications that were difficult to build before. We will discuss each of the three pieces, highlighting our goals and remaining work.

6.2.1 Investigation of Context

We detailed the results of our investigation of context in Section two. Our goal was to gain a better understanding of what context was and how it is used. This understanding will help us to choose what context to use in our applications and has provided insights into the types of data that need to be supported and the abstractions and mechanisms required to support context-aware computing.

In one sense, we have been quite successful. We have used our survey of context-aware computing to identify the current design process for building context-aware applications and to determine which of its activities were accidental and which were essential. This led to our new, simpler design process. Furthermore, our investigation of context helped us identify the context acquisition, interpretation, and aggregation abstractions, and the required and useful features for an architecture that supports context-aware applications.

However, we have not been so successful in using our understanding to generate a taxonomy of context. After several failed attempts (each ending in fruitful exercise to model the entire world), we believe the correct research path is to identify the orthogonal dimensions of context, such as spatial relationships (in front of, on top of, etc.) and hierarchy, temporal granularity (before, after, past and future), and entity granularity (group information), for example. An understanding of the dimensions of context will enable us to identify additional mechanisms for dealing with context (such as a geometry model for dealing with spatial relationships [Brumit99, Nelson98]) and identify context types that have not been widely used in context-aware applications (such as group context). This identification of the orthogonal dimensions of context has yet to start.

6.2.2 Implementation of the Architecture

The second piece of our proposed research is the implementation of the abstractions and mechanisms we identified and presented in Section five. This includes providing an implementation of the simpler design process and providing a design document that illustrates how designers should both think about building context-aware applications and how they can use the architecture to actually build them. Our research goals are to enable application designers to easily build and evolve context-aware applications by following the simpler design process. Additionally, we want to make the addition and removal of sensors transparent to running applications and to make the implementation of context acquirers or widgets [Salber99], context interpreters and context aggregators or servers a simple process.

We have already built a large portion of the architecture [Dey99c]. The main components and reusable building blocks in this architecture are context widgets, context interpreters and context servers. Context widgets implement the sensor abstraction, hiding the details of the sensor being used to acquire context. Each context widget is responsible for a single piece of context. Context interpreters implement the interpretation abstraction. Context servers implement the aggregation abstraction. They are context widgets that are responsible for an entity's entire context.

Through these components, all of the required features and most of the useful features have been implemented. They share a common communications mechanism that supports access to components and context on distributed computing devices. The communications protocol and language used is transparent to the components. Each of these components is autonomous in execution. They are instantiated independently of each other and execute in their own threads, supporting our requirement for independence. They can be instantiated all on a single computing device or on multiple computing devices. All of these components have the ability to query or subscribe to the context contained in context widgets and servers. Context widgets and servers automatically provide context storage and support for querying and subscriptions, so the designer does not have to deal with these issues.

We have multiple examples of context widgets and interpreters. We have built a few sample context servers, but there are some outstanding technical issues that still need to be resolved. Support for resource discovery has not yet been added to the architecture. This is the most complex feature and will require the most time for implementation.

All the components were designed from the beginning to be extremely flexible. The communications protocol and language can be replaced easily, as can the context storage mechanism. We currently use the HyperText Transfer Protocol (HTTP) and the eXtensible Markup Language (XML) for our default communications, and the mySQL database for storing context. We have not yet built alternatives and actually replaced the defaults, although we believe this to be a simple task. The components were designed to use common mechanisms that can be found in many programming languages and implemented on many

platforms. For example, the default communications mechanism that uses HTTP and XML only requires that a programming language support text parsing and TCP-IP communications, features common in most languages and available on most platforms.

It is important to point out that the particular implementation choices we make (or have already made) are not important. We could use existing off-the-shelf components like CORBA (Common Object Request Broker Architecture) [OMG91] or Jini [Arnold99] or research solutions like Linda [Gelernter85] or the Open Agent Architecture [Cohen94] to implement the abstractions, requirements and features we have identified. These or other implementations can be added or replaced through the flexible design of the architecture. What is important is our hypothesis that this collection of architecture features and abstractions supports our simpler design process and makes it easy for application developers to build and evolve their context-aware applications.

There are many features of the architecture that require validation. Some of the validation work has been performed at this time, through the design and implementation of example applications. A simple application such as the In Out Board can demonstrate the architecture's support for the following features:

- support for distribution of context and sensing;
- support event management;
- support independence and availability of components that collect context;
- support for the interpretation of context;
- support for transparent communications; and,
- support for using and building of abstractions.

A more complex application such as the Conference Assistant [Dey99d] (to be described in the next sub-section) is required to demonstrate the application's support for:

- support the storing of context history, and
- support for the aggregation of context information.

To demonstrate the final two features of the architecture, flexibility/pluggability and the use of language and platform-independent mechanisms, more work needs to be done. For the first feature, sample applications that provide their own communications and storage implementations can be implemented. For the second feature, an implementation of the architecture in a second programming language will suffice. A partial implementation of the architecture has been performed in Frontier and C++, allowing applications to be written in both of those languages. Currently, we (and others) are working on a full implementation of the architecture in C++ and Squeak, to complement our original implementation in Java. We have already demonstrated that the architecture executes on multiple platforms including UNIX, Windows 95/NT, Windows CE, and Macintosh.

To validate the ease in which the design process can be followed, new applications built, old applications evolved, and context components implemented, we intend to provide the architecture to other students (through the Hackfest class and the Aware Home project [Kidd99]) to use. We would like to compare the end products of multiple application developers building the same application with our design process, to see whether we have made the process as simple as possible. We would also like to analyze the time and the number of lines of code required to develop and implement new and evolving applications. Finally, we will use anecdotal feedback to determine whether there is additional support we can provide to make use of the architecture easier. We have some evidence that the architecture is easy to use. Members of our research group have successfully used the architecture in a timely fashion to build and evolve some context-aware applications.

6.2.3 Advanced Applications and Research

The third portion of a research is to investigate the architecture's usefulness as a research platform that can be used to examine difficult problems in context-aware computing and to build advanced applications. To illustrate this, we propose to do a study of group context. Group context involves multiple entities whose context is required simultaneously to perform some action. This area of context-aware computing has been relatively untouched due to the difficulty in building complex context-aware applications. We intend to

study existing research on group interactions to identify some generic mechanisms that are useful when dealing with group context. If necessary, we will also perform an ethnographic study of one of the following groups: C2000 research group, CNS, a family in a home setting. We will implement the identified mechanisms on top of the general context mechanisms already provided in the architecture. There are two goals for our investigation of group context. The first is to show how the architecture can be used to as an open framework on which detailed explorations of advanced topics in context-aware computing can be performed. The second goal is to demonstrate that the architecture simplifies the design of complex context-aware applications such as those involving group context.

We have built a number of simple applications with the architecture. In [Salber99], we discussed an In Out Board, an Information Display, and an informal meeting capture system. The In Out Board keeps track of occupants of a building, determining whether they are in or out of the building and when they were last seen. It uses identity, time and location context. A web version additionally uses the location of the person viewing the board to modify the display. Viewers on the Georgia Tech campus are shown all of the information, whereas viewers off-campus are shown only in/out status with no time information. The Information Display is an application that displays information relevant to the user that approaches it. It uses identity, location, and personal "profile" context. The informal meeting capture system, DUMMBO [Brotherton98], is an whiteboard that can capture what is written on it and the audio signal created around it. When multiple people are around DUMMBO, it begins the recording process. When those people leave, the recording process is stopped. At a later time, the identities of these people and the time of the recording could be used to retrieve the captured information, as a form of context-based retrieval [Lamming94]. DUMMBO uses identity, time, and location information.

In addition to these applications, we have built a context-aware mailing list that has knowledge of who is in a particular building. When users enter the building, they are added to the mailing list, and when they leave the building, they are removed from the list. This allows other users to send an email message to a single consistent email address and have it be received by a dynamically changing group of people, the current occupants of the building. This application uses only identity and location as context.

A more advanced or complex application that we have built is a prototype context-aware tour guide application. When a user starts this application, she is asked for a list of personal interests. These interests are used to generate a set of interesting sites to visit on the tour. When the user visits a site, she is shown information relevant to that site. She can indicate her level of interest with the site, and this dynamic information is used to update the list of potentially interesting and unvisited sites. After the tour is over, a trip report is emailed to the user. This contains information about each site the user visited, including name of the site, description of the site, time of visit, web address to obtain more information from, and the user's level of interest in the site. This tour guide application uses location, identity, time, and personal preferences as context.

Our most complex application is the prototype Conference Assistant [Dey99d]. This application aids a conference attendee in determining what presentations to see at a conference, locating her colleagues and taking notes on presentations. When a user arrives at a conference, he is given a Personal Digital Assistant (PDA), and asked to enter a list of research interests and the names of his colleagues. The PDA then displays a schedule for the conference, highlighting the presentations that are potentially interesting to the user (based on his interests) and indicating the locations of his colleagues, if known. When the user enters a presentation room, the PDA displays the title of the presentation, the name of the presenter, and a thumbnail image of the current PowerPoint slide or web page being presented. As the presentation continues, the display is updated accordingly. The user can enter notes on the current or previous slides and indicate his level of interest in the presentation. The level of interest information is shared with his colleagues, just as theirs is displayed to him. In the case where the user is attending a presentation that is not interesting to him, this information is useful for determining which presentation he may want to move to. After the conference, the user can use context-based retrieval to retrieve information about the conference, including his personal notes on the presentations he attended, as well as any presented information. The user can retrieve this information using the following as indices: keyword, user or colleague attendance, user or colleague question, or research interests. This application would be difficult to build without the context-aware architecture. It uses similar context (identity, location, time, personal

preferences and presented information) as the other described applications, but what makes it complex is the amount of context used, the dynamic nature of the context, and the likelihood of simultaneous context updates.

We will continue to build complex applications such as these. In particular, we aim to build applications that push on many dimensions of scale including time (available 24 hours a day, 7 days a week), space (available in multiple locations), and number of people (simultaneous use by multiple people). For this reason, we chose to investigate group context. We have not yet begun our study of group context, but we have identified one general mechanism that we will need to support. This mechanism supports comparison between multiple entities. One common feature in group applications is to create a sub-grouping based on a common piece of information. For example, the people in a given room form an informal group, as do the people who share a common interest. A general comparison mechanism will ease the development of group context applications that use this type of feature. We believe that in our study of group context, we will find additional general mechanisms to support.

6.3 Expected Contributions

There are three expected contributions of this research, one for each of the proposed pieces of research discussed in the previous sub-section. The first is contribution is an intellectual one. By providing both the orthogonal dimensions of context and 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 infrastructures will inform other infrastructure builders in building their own solutions.

The second contribution of our research is to lower the threshold for application designers trying to build context-aware applications. 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 have that will be implemented in our architecture handles the 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 architecture will use lightweight integration mechanisms allowing for the easy addition of context to non-context-aware applications.

The third contribution of our research is to raise the ceiling in terms of what researchers can accomplish in context-aware computing. The context architecture will allow 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 now 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 multiple locations, multiple people, always available, with simultaneous and independent activity.

7 Timetable for Completion

In this section, we will provide a timeline for completing the remaining work that was outlined in the previous section.

Task	Complete By
Part 1: Investigation of Context	
Identification of the orthogonal dimensions of context	March 2000
Part 2: Implementation of the Architecture	
Implementation of resource discovery	October 1999

Validation of flexibility/pluggability	October 1999
Validation of language-independent mechanisms	December 1999
Provide architecture to students with plans for testing ease of use	September 1999
Study ease of use	March 2000
Part 3: Advanced Applications and Research	
Investigation of group context	March 2000
Implementation of example group context applications	September 2000
Part 4: Dissertation	
Defense of Research	December 2000
Submission of Thesis	March 2001

Bibliography

Abowd97a: Gregory D. Abowd, Chris G. Atkeson, Jason Hong, Sue Long, Rob Kooper & Mike Pinkerton, Cyberguide: A mobile context-aware tour guide, ACM Wireless Networks, 3(5), 1997, pp. 421-433. (*cited on page 11*)

Abowd97b: Gregory D. Abowd, Anind K. Dey & Andy Wood, Applying dynamic integration as a software infrastructure for context-aware computing, Georgia Tech Technical Report, GIT-GVU-97-18, September 1997. (*cited on page 14*)

Abowd99: Gregory D. Abowd, Anind K. Dey, Jason Brotherton & Robert J. Orr, Context-awareness in wearable and ubiquitous computing, Virtual Reality 3, 1999, pp. 200-211. (*cited on page 15*)

Adly97: Noha Adly, Pete Steggles & Andy Harter, SPIRIT: A resource database for mobile users, in Proceedings of the CHI '97 Workshop on Ubiquitous Computing, March 1997. (*cited on page 9*)

Arnold99: Ken Arnold, Bryan O'Sulivan, Robert W. Scheifler, Jim Waldo & Ann Wolrath, The Jini specification, 1st edition, Addison-Wesley Publishing Company, June 1999. (*cited on page 24*)

Bauer98: Martin Bauer, Timo Heiber, Gerd Korteum & Zary Segall, A collaborative wearable system with remote sensing, in Proceedings of 2nd International Symposium on Wearable Computers, ISWC '98, October 1998, pp. 10-17. (*cited on page 9*)

Brooks87: F.P. Brooks, No silver bullet: Essence and accidents of software engineering, IEEE Computer, 20(4), April 1987, pp. 10-19. (*cited on page 6*)

Brotherton98: Jason A. Brotherton, Gregory D. Abowd & Khai N. Truong, Supporting capture and access interfaces for informal and opportunistic meetings, Georgia Tech Technical Report, GIT-GVU-99-06, December 1998. (*cited on page 25*)

Brown96: Peter J. Brown. The stick-e document: A framework for creating context-aware applications, in Proceedings of EP '96. (*cited on page 13*)

Brown97: Peter J. Brown, John D. Bovey & Xian Chen, Context-aware applications: From the laboratory to the marketplace, IEEE Personal Communications, 4(5), October 1997, pp. 58-64. (*cited on page 14*)

Brumitt99: Barry Brumitt, John Krumm, Brian Meyers & Steven Shafter, EasyLiving: Ubiquitous computing and the role of geometry, Position paper for the Inter-Agency Workshop on Research Issues for Smart Environments, July 25-26, 1999. (*cited on page 23*)

Clark91: H.H. Clark & S.E. Brennan, Grounding in communication, in L.B. Resnick, J. Levine, & S.D. Teasley (editors.), Perspectives on socially shared cognition, Washington, DC. 1991. (*cited on page 2*)

Cohen94: Philip R. Cohen, Adam Cheyer, Michelle Wang & Soon Cheol Baeg, An open agent architecture, in Proceedings of AAAI Symposium Series on Software Agents, March 1994, pp. 1-8. (*cited on pages 13, 24*)

Cooperstock97: Jeremy Cooperstock, Sidney Fels, William Buxton & K. Smith, Reactive environments: Throwing away your keyboard and mouse, Communications of the ACM 40(9), 1997, pp. 65-73. (*cited on page 12*)

Davies97: Nigel Davies, S.P. Wade Adrian Friday & G.S. Blair, Limbo: A tuple space based platform for adaptive mobile applications, in Proceedings of International Conference on Open Distributed Processing/Distributed Platforms, ICODP/ICDP '97, May 1997. (*cited on page 12*)

Dey97: Anind K. Dey, Gregory D. Abowd, Mike Pinkerton & Andy Wood, CyberDesk: A framework for providing self-integrating ubiquitous software services, Georgia Tech Technical Report, GIT-GVU-97-10, April 1997. Presented as a demonstration in the Proceedings of User Interface Software and Technology, UIST ’97, November 1997, pp. 75-76. (*cited on page 14*)

Dey98a: Anind K. Dey, Gregory D. Abowd & Andy Wood, CyberDesk: A framework for providing self-integrating context-aware services, in Proceedings of Intelligent User Interfaces, IUI ’98, January 1998, pp. 47-54. (*cited on page 14*)

Dey98b: Anind K. Dey, Context-aware computing: The CyberDesk Project, in Proceedings of AAAI Spring Symposium on Intelligent Environments, AAAI Technical Report SS-98-02, March 1998, pp. 51-54. (*cited on page 14*)

Dey99a: Anind K. Dey, Gregory D. Abowd & Andy Wood, CyberDesk: A framework for providing self-integrating context-aware services, Knowledge-Based Systems 11, 1999, pp. 3-13. (*cited on page 14*)

Dey99b: Anind K. Dey & Gregory D. Abowd, Towards a better understanding of context and context-awareness, Georgia Tech Technical Report, GIT-GVU-99-22, June 1999. (*cited on pages 3, 9*)

Dey99c: Anind K. Dey, Daniel Salber, Masayasu Futakawa & Gregory D. Abowd. An architecture to support context-aware applications, Georgia Tech Technical Report, GIT-GVU-99-23, June 1999. (*cited on page 23*)

Dey99d: Anind K. Dey, Masayasu Futakawa, Daniel Salber & Gregory D, Abowd. The Conference Assistant: Combining context-awareness with wearable computing, To be presented at the 3rd International Symposium on Wearable Computers, ISWC ’98, October 1999. (*cited on pages 24, 25*)

DS98: Java iButtons™ manufactured by Dallas Semiconductor Incorporated, web page available at http://www.ibutton.com. (*cited on page 6*)

Friday96: Adrian J. Friday, Infrastructure support for adaptive mobile applications, Ph.D. Thesis, Lancaster University, September 1996. (*cited on page 13*)

Gelernter85: David Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 2(1), January 1985, pp. 80-112. (*cited on page 24*)

Harrison98: Beverly L. Harrison, Kenneth P. Fishkin, Anuj Gujar, Carlos Mochon & Roy Want, Squeeze me, hold me, tilt me! An exploration of manipulative user interfaces, in Proceedings of Conference on Human Factors in Computing Systems, CHI '98, April 1998, pp. 17-24. (*cited on page 11*)

Harter94: Andy Harter & Andy Hopper, A distributed location system for the Active Office, IEEE Network 8(1), January 1994, pp. 62-70. (*cited on page 9*)

Harter99: Andy Harter, Andy Hopper, Pete Steggles, Andy Ward & Paul Webster, The anatomy of a context-aware application, to be presented at MobiCom '99, August 1999. (*cited on page 9*)

Hull97: Richard Hull, Philip Neaves & James Bedford-Roberts, Towards situated computing, in Proceedings of 1st International Symposium on Wearable Computers, ISWC '97, October 1997. (*cited on page 15*)

Kidd99: Cory K. Kidd, Robert J. Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E. Starner & Wendy Newstetter, The Aware Home: A living laboratory for ubiquitous computing research, in Proceedings of the 2nd International Workshop on Cooperative Buildings, CoBuild '99, October 1999. (*cited on page 24*)

Korteum98: Gerd Korteum, Zary Segall & Martin Bauer, Context-aware, adaptive wearable computers as remote interfaces to 'intelligent' environments, in Proceedings of 2nd International Symposium on Wearable Computers, ISWC '98, October 1998, pp. 58-65. (*cited on pages 9, 13*)

Lamming94: Lamming, Brown, Carter, Eldridge, Flynn, Robinson, & Sellen, The design of a human memory prosthesis, Computer Journal 37(3), 1994, pp. 153-163. (*cited on page 25*)

Long96: Sue Long, Rob Kooper, Gregory D. Abowd & Christopher G. Atkeson, Rapid prototyping of mobile context-aware applications: The Cyberguide case study, in Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking, MobiCom '96, November 1996. (*cited on page 11*)

Mynatt98: Elizabeth D. Mynatt, Maribeth Back, Roy Want, Michael Baer & Jason B. Ellis, Designing Audio Aura, in Proceedings of Conference on Human Factors in Computing Systems, CHI '98, April 1998, pp. 566-573. (*cited on pages 9, 13*)

Nelson98: Giles J. Nelson, Context-aware and location systems, Ph.D. Thesis, University of Cambridge, January 1998. (*cited on pages 5, 15, 23*)

OMG91: Object Management Group, The common object request broker: Architecture and specification, Revision 1.1, OMG Document Number 91.12.1, December 1991. (*cited on page 24*)

Pascoe98: Jason Pascoe, Adding generic contextual capabilities to wearable computers, in Proceedings of 2nd International Symposium on Wearable Computers, ISWC '98, October 1998, pp. 92-99. (*cited on page 8, 9 ,15*)

Pederson97: E.R. Pederson & T. Sokoler, AROMA: Abstract representation of presence supporting mutual awareness, in Proceedings of Conference on Human Factors in Computing Systems, CHI '97, March 1997, pp. 51–58. (*cited on page 12*)

Rekimoto96: Jun Rekimoto. Tilting operations for small screen interfaces, in Proceedings of User Interface Software and Technology, UIST '96, November 1996, pp. 167-168. (*cited on page 11*)

Richardson95: Tristan Richardson, Teleporting – Mobile X sessions, in Proceedings of the 9th X Technical Conference, January 1995. (*cited on page 9*)

Rodden98: Tom Rodden, Keith Cheverst, Nigel Davies & Alan Dix, Exploiting context in HCI design for mobile systems, in Proceedings of Workshop on Human Computer Interaction with Mobile Devices, HCIMD '98, May 1998. (*cited on page 17*)

Salber99: Daniel Salber, Anind K. Dey & Gregory D. Abowd, The Context Toolkit: Aiding the development of context-enabled applications, in Proceedings of Conference on Human Factors in Computing Systems, CHI '99, May 1999, pp. 434-441. (*cited on page 18, 23, 25*)

Schilit94a: William N. Schilit, Norman I. Adams & Roy Want, Context-aware computing applications, in Proceedings of the 1st International Workshop on Mobile Computing Systems and Applications, December 1994, pp. 85-90. (*cited on page 9*)

Schilit94b: William N. Schilit & Marvin Theimer, Disseminating Active Map information to mobile hosts, IEEE Network, 8(5), September/October 1994, pp. 22-32. (*cited on page 4*)

Schilit95: William N. Schilit, System architecture for context-aware mobile computing, Ph.D. Thesis, Columbia University, May 1995. (*cited on pages 9, 14*)

Schmidt98: Albrecht Schmidt, Michael Beigl & Hans-Werner Gellersen, There is more to context than location, in Proceedings of Interactive Applications of Mobile Computing, IMC'98, November 1998. (*cited on page 9*)

Schwartz92: M.F Schwartz *et al*., A comparison of internet resource discovery approaches, Computing Systems, Fall 1992, pp. 461-493. (*cited on page 18*)

Ullmer97: Brygg Ullmer & Hiroshi Ishii., The metaDESK: Models and prototypes for tangible user interfaces, in Proceedings of User Interface Software and Technology, UIST '97, October 1997. (*cited on page 12*)

Want92: Roy Want, Andy Hopper, Veronica Falcao & Jonathan Gibbons. The Active Badge location system, ACM Transactions on Information Systems 10(1), January 1992, pp. 91–102. (*cited on pages 8, 12, 17*)

Want95: Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis & Mark Weiser, An overview of the PARCTab ubiquitous computing experiment, IEEE Personal Communications, 2(6), December 1995, pp. 28-43. (*cited on page 8*)

Ward98: Andrew M.R. Ward, Sensor-driven computing, Ph.D. Thesis, University of Cambridge, August 1998. (*cited on page 9*)

Weiser91: Mark Weiser. The computer for the 21st century, Scientific American 265(3), September 1991, pp. 94-104. (*cited on page 2*)

Weiser96: Mark Weiser & John S. Brown. Designing calm technology, PowerGrid Journal 1.01, July 1996. (*cited on page 2*)

Whitehead99: E. James Whitehead, Jr., Rohit Khare, Richard N. Taylor, David S. Rosenblum & Michael M. Gorlick. Architectures, protocols, and trust for info-immersed active networks, Position paper for the Inter-Agency Workshop on Research Issues for Smart Environments, July 25-26, 1999. (*cited on page 6*)