Spatial Connector
Loosely Binding Contextual Changes and Non-Context-Aware Services

Ichiro Satoh
National Institute of Informatics, 2-1-2 Hitotsubashi Chiyoda-ku Tokyo, 101-8430, Japan

Keywords: Context-awareness, Software Reusability, Ubiquitous Computing, Separation of Concerns.

Abstract: A framework for providing context-aware services is presented. Context-aware services tends to depend on context but software for defining the services should be independent on context so that it can be reused in other contexts. However, software for many context-aware services in existing approaches has been constructed in an ad-hoc manner. The approach, called *Spatial Connector*, enables software for context-aware services to be defined independently of any contextual information so that it can be reused in other context. It enables non-context-aware services to be used as context-aware services by deploying services according to contextual changes and transforming such changes into non-context-aware functions, so that software for context-aware services to be defined independently on any contextual information. Our early experiments proved that it enabled us to reuse JavaBeans components as context-aware services without having to modify the components themselves.

1 INTRODUCTION

Context-aware computing means that application-specific services are aware of and depend on the underlying runtime contexts and context changes. Here contexts can be generally defined as external varying environments that affect or determine the computation of an application. Examples of such environments or contexts include time, location, situation, and users. By being aware of the context, a system can be proactive or reactive towards users by providing information or services the user needs in a particular context. Instead of a tool, technologies evolve into pragmatic systems which support our daily lives. Software for context-aware services directly or indirectly specify how computations or behaviors of context-aware services depend on or vary in underlying contexts.

Context-aware services often result in software engineering problems. Context-aware services themselves must depend on particular context. However, software for context-aware services often tends to have been developed dependently on such context. This prevents context-aware services from being reused for other context. In fact, some software for location-aware annotation services may include information about the locations that the services are provided. This is because it is difficult to abstract contextual information from software in comparison with other information.

Furthermore, software for most context-aware services has been developed independently of those for web services and enterprise-services, although the former services are similar to the latter services. Therefore, we cannot reuse software for web services and enterprise-services as context-aware services.

We need a software engineering approach, called *Spatial Connector*, to reuse software for context-aware services. This paper proposes a mechanism for loosely coupling between software for (non-context-aware) services and contexts. The key idea behind our mechanism is to define contextual information outside software for services and loosely link between services and contextual information measured by the underlying sensing systems. In fact, the approach enabled us to reuse JavaBeans components for Web services as context-aware services without modifying the components themselves.

2 RELATED WORK

Since the term *context-aware computing* was introduced by Schilit and Theimer (Schilit and Theimer, 1994), context-aware computing received a lot of attention from researchers. Many researches have stud-

Satoh I..
Spatial Connector - Loosely Binding Contextual Changes and Non-Context-Aware Services.
DOI: 10.5220/0004494500500057
In *Proceedings of the 8th International Joint Conference on Software Technologies* (ICSOFT-EA-2013), pages 50-57
ISBN: 978-989-8565-68-6

ied software engineering for context-aware services. The Context toolkit was a pioneer work of software engineering issues in context-aware services (Abowd, 1999; Salber et al., 1999). It aimed at allowing programmers to leverage off existing building blocks to build interactive systems more easily. It was constructed as libraries as widgets for GUI. However, since it was designed only for context-aware services, it did not support the reuse of software for non-context-aware services.

Ubiquitous computing defines a new domain in which large collections of heterogeneous devices are available to support the execution of applications. These applications become dynamic entities with multiple input and output alternatives. As a result, it is difficult to predict in advance the most appropriate application configuration as discussed by several researchers (Roman et al., 2003; Scholtz et al., 2004). There have been many attempts to construct software component technology for ubiquitous computing (Flissi et al., 2005; Modahl et al., 2005). Several researchers have studies modeling of context-awareness in the literature of software engineering (Henricksen and Indulska, 2005). However, there have been no silver bullet as other systems so far.

Several researchers focuses on high-level analysis on context-aware services. For example, A comparable model, which specify context-aware behavior in only one diagram, the context-oriented domain analysis diagram was explored (Desmet et al., 2007). There have been several attempts to specify context-aware services (Henricksen et al., 2002). However, they cannot solve problems in software-level dependencies with contexts.

We developed a mobile agent-based emulator to emulate the physical mobility of its target terminal by using the logical mobility of the emulator. It could provide application-level software with the runtime environment compatible to its target device and carry it between computers through networks. However, it intends to make it easy to test context-aware software rather to develop such software.

Modern enterprise architectures, e.g., Enterprise JavaBeans (EJB), (Kassem, 2000) and .Net architecture (Szyperski, 1998), have employed the notion of container to separate the business components from the system components. The original notion has enabled key functionality such as transactions, persistence, or security to be transparently added to the application at deployment time rather than having to implement it as part of the application. The notion leads to increased reusability and interoperability of business components. We use the notion to reuse non-context-aware business software components in context-aware ones. Non-context-aware components are not designed to be used in ubiquitous computing environments, where services appear and disappear arbitrarily and nodes cannot possibly know in advance with which other nodes they will interact.

3 APPROACH

This section outlines our approach.

3.1 Example Scenario

To enable services that enhance user interaction with his/her current environment, we need to enrich his/her physical surrounding, e.g., shopping malls, museums, and trade fares, with dedicated computing resources that enable service to be provided. One example scenario is a shopping mall that offers ambient services to customers, enabling them to navigate through the mall and find certain products quickly. Users moving from shop to shop should have their services deployed and executed at stationary terminals close to them to support them. Annotation services on appliances, e.g., electric lights, may be provided in shops. Such services depend on context. For example, they present digital signage for sales promotions while their target appliances are displayed on shelves. For example, shops frequently replace and relocate their products inside them. The services need to follow the movement of their targets. Annotation services in appliances are needed not only at shops to explain what the appliances are but at users' homes to explain how they are to be used. Furthermore, such annotation services may depend on users. Some may want audio annotations but others may want visual annotations.

3.2 Design Principles

The *Spatial Connector* approach introduces the following three novel elements:

- *Container* is introduced to reuse existing non-context-aware software components, e.g., JavaBeans, as components for providing context-aware service. Each service container is a runtime environment that manages the execution of at most one component and invokes specified methods defined in the component according contextual changes in the real world by using sensing systems.
- *Counterpart* is a reference to its target, e.g., a user, physical entity, or computing device and contains profiles about the target, e.g., the name of the user,

the attributes of the entity and the network address of the device. It always deploys at computers close to the the current location of its target by using location-sensing systems.

- *Connector* is used to enable services to be dynamically deployed at computers according to contextual changes, e.g., the movements of users, physical entities, and devices. It defines a spatial relationship between containers and the counterparts of the targets that the services should be provided for. When a counterpart is deployed at another computer, it dynamically deploys one or more containers to certain computers, e.g., computers that contains the counterpart.

Like container technologies for enterprise computing, the first supports transformations that extend the functionality provided by the components in isolation. While a container may impose architectural constraints on the components it hosts, the functionality extensions provided by the container can often be achieved without the hosted components having been explicitly designed to support them. The first supports separation of concerns in the functions of components, the second abstracts away the underlying location sensing systems, and the third supports separation of concerns in the locations of components.

We here explain the reason why services need to be dynamically deployed. Computing devices in ambient computing environments may only have limited resources, such as restricted levels of CPU power and amounts of memory. They cannot support all the services that may be needed. We therefore have to deploy software that can define services at computers only while those services are needed. The deployment of services at computing devices does not only depend on the requirements of the services. For example, if a user has a portable computer, his/her services should be provided from the portable computer. Otherwise, such services should be provided from stationary computers close to him/her, even when the services may be initially designed to run on portable computers.

4 DESIGN AND IMPLEMENTATION

Our user/location-aware system to guide visitors is managed in a non-centralized manner. It consists of four subsystems: 1) location-aware directory servers, called LDSs, 2) service runtime systems, 3) *counterparts*, and 4) *containers*. The first is responsible for reflecting changes in the real world and the locations of users when services are deployed at appropriate computers. The second runs on stationary computers located at specified spots close to exhibits in a museum. It can execute application-specific components via containers, where we have assumed that the computers are located at specified spots in public spaces and are equipped with user-interface devices, e.g., display screens and loudspeakers. It is also responsible for managing *connectors*. The third is managed by the first and deployed at a service runtime system running on a computer close to its target, e.g., person, physical entity, or space. The fourth is implemented as a mobile agent. Each mobile agent is a self-contained autonomous programming entity. Application-specific services are encapsulated within the fourth.

The system has the three unique functions as follows:

- The *counterpart* is a digital representation of a user, physical entity, or computing device. When its target moves to another location, it is automatically deployed at a computer close to the current location of the target by using location-sensing systems.
- The *container* is a customizable wrapper for (non-context-aware) software components, e.g., JavaBeans, for defining application-specific services to use them as context-aware services.
- The *connector* is a relationship between the locations of at least one counterpart and service-providers. It enables services to be dynamically deployed at computers according to spatial changes in their targets.

The counterpart is responsible for abstracting away differences between the underlying location sensing systems. Each container can contain at most one application-specific component, e.g., JavaBeans component. It is responsible for its inner component with its favorite runtime environment. Therefore, such a component can be executed in our system without modifying it. Each container can explicitly specify at most one *connector*. The current implementation provides two types of *connectors*, as shown in Figure 1.

- If a container declares a *follow* connector for at most one counterpart, when the latter migrates to a computer in another location, the former migrates to the same or another computer in the latter's destination location.
- If a container declares a *shift* connector for at most one counterpart, when the latter migrates to a computer in another location, the former migrates

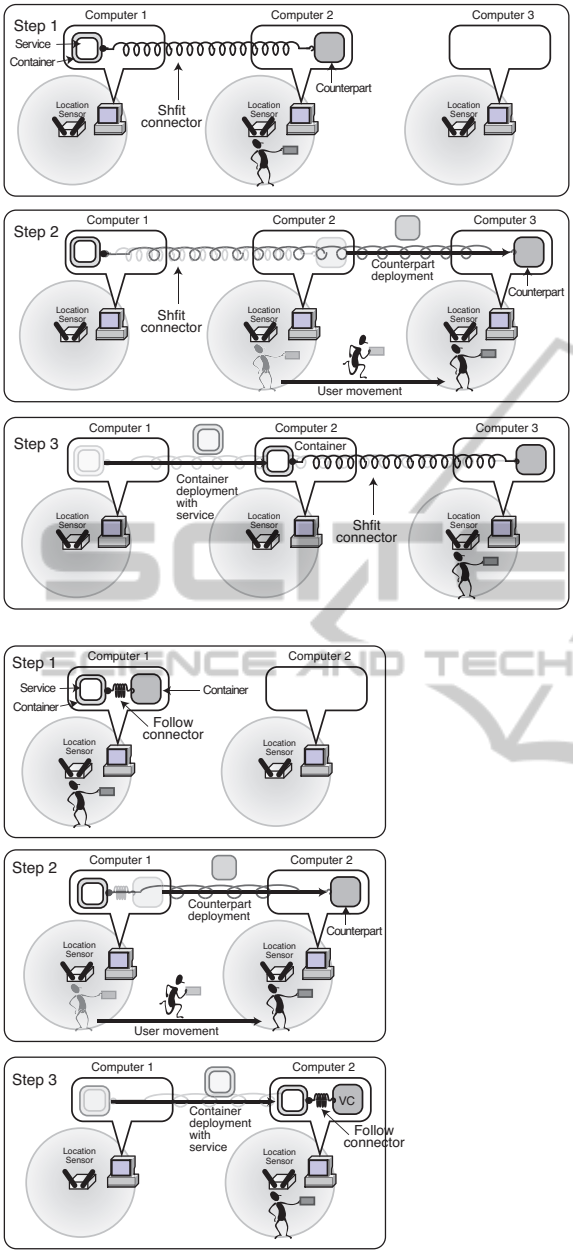

Figure 1: Spatial coupling between counterparts and containers.

to the latter's source or another computer in the latter's source location.

These can be dynamically bound between counterparts and service-providers. By using these relations, containers are independent of their locations and their deployment policies. When a user is in front of a product, his/her counterpart is deployed at a computer close to his/her current location by using a location-sensing system. Containers that declare follow policies for the counterpart are deployed at the computer. That is, our containers should accompany their users and annotate exhibits in front of them in the real-world. Nevertheless, users and containers are loosely coupled, because the containers are dynamically linked to the counterparts corresponding to the users.

4.1 Context Management

Each counterpart is attached to at most one target, e.g., a user, physical entity, or space. Each counterpart keeps the identifier of its target or RFID tag attached to the target. Each LDS is responsible for monitoring location-sensing systems and spatially binding more than one counterpart to each user or physical entity (Fig. 2). It maintains two databases. The first stores information about all the service runtime systems and the second stores all the containers attached to users or physical entities. It can exchange this information with other LDSs in a peer-to-peer manner. Each LDS only maintains up-to-date information on partial contextual information instead of on tags in the whole space. All LDSs and service runtime systems periodically multicast their network addresses to other LDSs and service runtime systems through UDP-multicasting. Therefore, when a new LDS or service runtime system is dynamically added to or removed from the whole system, other systems are aware of changes in their network domains.

Location-sensing systems can be classified into two types: proximity and lateration. The first approach detects the presence of objects within known spots or close to known points, and the second estimates the positions of objects from multiple measurements of the distance between known points. The current implementation assumes that museums have provided visitors with tags. These tags are small RF transmitters that periodically broadcast beacons, including the identifiers of the tags, to receivers located in exhibition spaces. The receivers locate the presence or positions of the tags. To abstract away differences between the underlying location-sensing systems, the LDSs map geometric information measured by the sensing systems to specified areas, called *spots*, where the exhibits and the computers that play the annotations are located.

4.2 Counterpart

Each counterpart is automatically deployed at a service runtime system running on a computer in a spot that contains its target by LDSs. We explain how to deploy counterparts according to changes in the real world. When the underlying sensing system de-

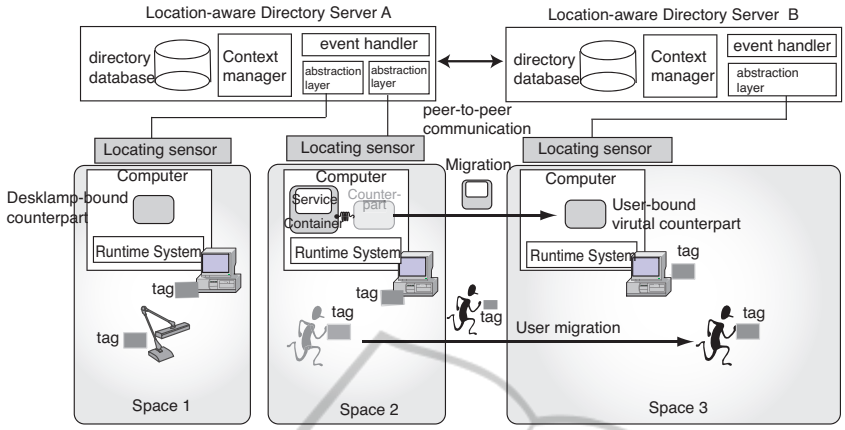

Figure 2: Location-aware directory server.

tects the presence (or absence) of a tag in a spot, the LDS that manages the system attempts to query the locations of the counterpart tied to the tag from its database. If the database does not contain any information about the identifier of the tag, it multicasts a query message that contains the identity of the new tag to other LDSs through UDP multicasting. It then waits for reply messages from the other LDSs. Next, if the LDS knows the location of the counterpart tied to the newly visiting tag, it deploys the counterpart at a computer close to the spot that contains the tag.

4.3 Service Runtime System

Each service runtime system is responsible for executing and migrating application-specific components with containers to other service runtime systems running on different computers through a TCP channel using mobile-agent technology. It is built on the Java virtual machine (Java VM version 1.5 or later), which conceals differences between the platform architectures of the source and destination computers. It governs all the containers inside it and maintains the life-cycle state of each application-specific component via its container. When the life-cycle state of an application-specific component changes, e.g., when it is created, terminates, or migrates to another runtime system, its current runtime system issues specific events to the component via its container, where the container may mask some events or issue other events.

4.4 Container

Each container is an autonomous programmable entity implemented as a mobile agent. Containers can provide application-specific components with their favorite runtime environments and carry them between computers. They are defined according to types of application-specific components. Each container in the current implementation is a collection of Java objects and support Java-based components, e.g., JavaBeans and Java Applets. It can migrate from computer to computer and duplicate itself by using mobile agent technology.[1] When a component is transferred over the network, not only the code of the agent but also its state is transformed into a bitstream by using Java's object serialization package and then the bit stream is transferred to the destination. Since the package does not support the capturing of stack frames of threads, when an agent is deployed at another computer, its runtime system propagates certain events to instruct it to stop its active threads. The runtime system on the receiving side receives and unmarshals the bit stream. Arriving agents may explicitly have to acquire various resources, e.g., video and sound, or release previously acquired resources.

4.5 Connector

The *Spatial Connector* approach enables users to explicitly assign a *connector* to each container through its management GUI, even while the container and its inner component are running. Each connector is activated when its target counterpart migrates to a computer in another cell due to the movement of the counterpart's target in the physical world. We will explain how to deploy containers, including its inner components with connectors according to the deployment of the counterparts that are specified in the connectors as their targets. The deployment of each container is specified in its connector and is managed by

[1] JavaBeans can easily be translated into agents in this platform.

runtime systems without any centralized management system. Each runtime system periodically advertises its address to the others through UDP multicasting, and these runtime systems then return their addresses and capabilities to the runtime system through a TCP channel. The procedure involves four steps. 1) When a counterpart migrates to another runtime system (in a different cell), 2) The destination sends a query message to the source of the visiting counterpart. 3) The source multicasts a query message within current or neighboring sub-networks. If a runtime system has a container whose connector specifies the visiting counterpart, it sends the destination information about itself and its neighboring runtime systems. 4) The destination next instructs the container to migrate to one of the candidate destinations recommended by the target, because this platform treats every container as an autonomous entity.

5 CURRENT STATUS

A prototype implementation of this approach was constructed with Sun's Java Developer Kit, version 1.5 or later version. Although the current implementation was not constructed for performance, we evaluated the migration of a container based on connectors. When a container declares a *follow* or *shift* connector for a counterpart, the cost of migrating the former to the destination or the source of the latter after the latter has begun to migrate is 88 ms or 85 ms, where three computers over a TCP connection is 32 ms.[2] This experiment was done with three computers (Intel Core 2 Duo 2 GHz with MacOS X 10.6 and Java Development Kit ver.6) connected through a Fast Ethernet network. Migrating containers included the cost of opening a TCP-transmission, marshalling the agents, migrating them from their source computers to their destination computers, unmarshaling them, and verifying security.

Support for location-sensing systems: The current implementation supports two commercial tracking systems. The first is the Spider active RFID tag system, which is a typical example of proximity-based tracking. It provides active RF-tags to users. Each tag has a unique identifier that periodically emits an RF-beacon (every second) that conveys an identifier within a range of 1-20 meters. The second system is the Aeroscout positioning system, which consists of four or more readers located in a room. These readers can measure differences in the arrival timings of WiFi-based RF-pulses emitted from tags and estimate the positions of the tags from multiple measurements of the distance between the readers and tags; these measurement units correspond to about two meters.

Security To prevent malicious containers or application-specific components from being passed between computers, each runtime system supports a Kerberos-based authentication mechanism for agent migration. It authenticates users without exposing their passwords on the network and generates secret encryption keys that can be selectively shared between parties that are mutually suspicious. Since it can inherit the security mechanisms provided in the Java language environment, the Java VM explicitly restricts containers so that they can only access specified resources to protect computers from malicious containers or components.

6 APPLICATION: SUPPORT TO LIFECYCLE SUPPORT OF PRODUCTS

We experimented on and evaluated a context-aware annotation service for appliances, e.g., electric lights. This is unique to other existing active content for digital signage because it does not support advertising of its target appliance but assists users with controlling and disposing of the appliance. We attached an RFID tag to an electric light and provided a counterpart and connected the counterpart and containers for the target by using *connectors* according the locations of the light, e.g., a warehouse, store, and home. These containers had JavaBean objects as application-specific services inside them. They supported the lifecycle of the light from shipment, showcasing, assembly, usage, and disposal.

In warehouse

While the light was in the warehouse, its counterpart was automatically deployed at a portable terminal close to the light. Two kinds of application-specific components were provided for the experiment. The first was attached to the counterpart through the *follow* connector. This notified a server in the warehouse of its specification, e.g., its product number, serial number, and date of manufacture, size, weight, serial number, the date of manufacture. The second was attached to the counterpart through the *shift* connector and ordered more lights. The both two application-specific components themselves were not context-aware, because they were implemented as JavaBean objects. In

[2]The size of each counterpart was about 8 KB in size.

fact, we could reuse JavaBean objects running on a server with Java 2 Enterprise Edition to advertise the light from Web without modifying the objects themselves.

In store

While the light was being showcased in a store, we assumed that it had two application-specific components. The first declared the *follow* connector and was deployed at a computer close to its target object so that it displayed advertising content to attract purchases by customers who visited the store. Figures 3 a) and b) have two images maintained in the component that display the price, product number, and manufacture's name on the current computer. The second declared the *shift* connector. When the light was sold, it notified the warehouse server that the light was out of stock.

In house

When the light was bought and transferred to buyer's house, a container, which was attached to the counterpart through the *follow* connector and had an application-specific component inside it, migrated to a computer in the house and provided instructions on how it should be assembled. Figure 3 c) has the active content for advice on assembly. The component also advised how it was to be used as shown in Fig. 3 d). When it was disposed of, the component presented its active content to give advice on disposal. Figure 3 e) illustrates how the appliance was to be disposed of.

Figure 3: Digital signage for supporting appliance.

These application-specific components could be defined independently of any locations. This is useful to separate services in developing application-specific services.

We can provide application-specific component in buyers' houses that controls appliances, which may not have any network interfaces. The component allows us to use a PDA to remotely control nearby lights to use them as place-bound controller services. The services can communicate with X10-base servers to switch lights on or off and are indirectly attached to places with room lights in this system through their containers. Each user has a tagged PDA, which supports the runtime system with a wireless LAN interface. When a user with a PDA visits a spot that contains a light, the approach moves a controller component with its container to the runtime system of the visiting PDA. The component, now running on the PDA, displays a graphical user interface to control the light. When the user leaves that location, its container carried the component automatically closes its user interface and returns to its home runtime system.

7 CONCLUSIONS

We constructed an approach, called *Spatial Connector*, for binding non-context aware services with contexts. It supported the separation of services and context, so that application-specific services could be defined independently of any contextual information. It also provides three elements, called *container*, *counterpart*, and *connector*. The first supports separation of concerns in the functions of components, the second abstracts away the underlying location sensing systems, and the third supports separation of concerns in the locations of components. The approach enabled non-context-aware services to be used as context-aware services to be used as context-aware services. It enabled us to dynamically modify where, when, what, and how services should be activated.

The example presented in this paper was not expressively large-scale spaces and consequently did not have as many users as cities. Our final goal is a city-wide ubiquitous computing environment, which will provide a variety of services to massive numbers of users from numerous heterogenous computers. Therefore, our system itself is designed for a city-wide context-aware system.[3] In fact, Since the approach had no centralized management system, we believe that it was useful in city-wide context-aware services.

[3]It is almost impossible to experiment academic systems in city-wide spaces without any pre-evaluation in small-spaces.

REFERENCES

Abowd, G. D. (1999). Software engineering issues for ubiquitous compuitng. In *Proceedings of International Conference on Software Engineering (ICSE'99)*, pages 75–84. ACM Press.

Desmet, B., Vallejos, J., Costanza, P., De Meuter, W., and D'Hondt, T. (2007). Context-oriented domain analysis. In *Proceedings of the 6th international and interdisciplinary conference on Modeling and using context*, CONTEXT'07, pages 178–191. Springer-Verlag.

Flissi, A., Gransart, C., and Merle, P. (2005). A component-based software infrastructure for ubiquitous computing. In *Proceedings of the The 4th International Symposium on Parallel and Distributed Computing*, pages 183–190. IEEE Computer Society.

Henricksen, K. and Indulska, J. (2005). Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing, In*, 2:2005.

Henricksen, K., Indulska, J., and Rakotonirainy, A. (2002). Modeling context information in pervasive computing systems. In *Proceedings of the First International Conference on Pervasive Computing*, Pervasive '02, pages 167–180. Springer-Verlag.

Kassem, N. (2000). Designing enterprise applications with the java 2 plaform. Technical report, Sun Microsystems. Sun J2EE Blueprints, http://java.sun.com/j2ee/download.html.

Modahl, M., Agarwalla, B., Saponas, S., Abowd, G., and Ramachandran, U. (2005). Ubiqstack: a taxonomy for a ubiquitous computing software stack. *Personal Ubiquitous Computing.*, 10:21–27.

Roman, M., Al-muhtadi, J., Ziebart, B., Campbell, R., and Mickunas, M. D. (2003). System support for rapid ubiquitous computing application development and evaluation. In *Proceedings of Workshop on System Support for Ubiquitous Computing (UbiSys'03),5th International Conference on Ubiquitous Computing (UbiComp 2003*.

Salber, D., Dey, A. K., and Abowd, G. D. (1999). The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of International Conference on Computer-Human Interaction (CHI'99)*, pages 15–20. ACM Press.

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

Scholtz, J., Consolvo, S., Scholtz, J., and Consolvo, S. (2004). Towards a discipline for evaluating ubiquitous computing applications. Technical report, National Institute of Standards and Technology. [Online]. Available: http://www.itl.nist.gov/iad/vvrg/newweb/ubiq/docs/1 scholtz modified.pdf.

Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.