INTERFACE STRATEGY PATTERN FOR CONTEXT AWARE SERVICE BASED ENVIRONMENT

[illegible]

[illegible]

00001220

INTERFACE STRATEGY PATTERN FOR CONTEXT-AWARE SERVICE BASED ENVIRONMENT

Siphesihle Mandlakhe Zuma

A dissertation submitted to the Faculty of Science & Agriculture in fulfillment

of the requirements for the degree

MASTER OF SCIENCE

In

COMPUTER SCIENCE

Department of Computer Science

University of Zululand

2005

Declaration

I, ..., declare that this dissertation represents research work carried out by the author and has not been submitted in any form to another university for a degree. All sources that I have used have been duly acknowledged in the text.

Z 004.6 ZUM
0080C1220

Signature of the student

Dedication

To my mother, Philisiwe Witness Zuma

Contents

Declaration i

Dedication ii

Acknowledgements iii

Contents iv

List of Figures vii

List of Tables ix

Abstract xii

CHAPTER ONE INTRODUCTION 1

1.1 Introductory Background 1

1.2 Statement of the Problem 5

1.3 Research Goal and Objectives 6

1.3.1 Research Goal 7

1.3.2 Research Objectives 7

1.4 Research Methodology 7

1.5 Organisation of the Dissertation 8

CHAPTER TWO LITERATURE REVIEW 10

2.1 Introduction 10

2.2 Inter-component Communication Models 11

2.2.1 Dynamic Adjustment of Component InterActions (DACIA) 12

2.2.2 Scalable Timed Events And Mobility (STEAM) 13

2.2.3 Java Event-based Distributed Infrastructure (JEDI) 14

2.2.4 Gaia: Model for Developing Context-Aware Applications 16

2.2.5 The Hydrogen Context-Framework 19

2.3 The Proposed Solution Approach 20

2.4 Software Design Patterns 21

CHAPTER THREE THE PROPOSED MODEL 23

3.1 Introduction 23

3.2. The Context-Aware Component Interfacing 24

3.2.1 The Information Bus 29

3.2.2 A Common Message Format 29

3.2.3 Operations for Interaction 30

3.2.4 Retrieving Tuples from the Component Bus 33

3.2.5 Context-Awareness in CACIP 37

3.3 Dynamic Configuration Provisioning with CACIP 38

3.4 Comparison of CACIP with Existing Schemes 39

CHAPTER FOUR DESIGN AND IMPLEMENTATION OF THE PROTOTYPE..41

4.1 Introduction 41

4.2 Prototype Development for the Restaurant Locator Service (RLS) 42

4.3 Environment Specification 466

4.3.1 RLS User Interface 48

4.4 Performance Evaluation 51

4.4.1 Message Throughput 52

4.4.2 Round-trip Time 53

4.4.3 Role of Context-Awareness 54

CHAPTER FIVE CONCLUSIONS AND FUTURE WORK 57

5.1 Conclusions 57

5.2 Future Work 59

REFERENCES 61

List of Figures

Figure 1.1 An m-Commerce Product Line reference Architecture..................2

Figure 1. 2 Class Diagram for the Mobile Commerce Product Line Architecture...5

Figure 2.1 A logical view of JEDI..15

Figure 2.2 Class Diagram for the JEDI Architecture.................................16

Figure 2.3 Gaia Context Infrastructures 18

Figure 2.4 Class Diagram for the Gaia Context Infrastructure.....................19

Figure 3.1 The Context-Aware Component Interfacing Pattern – showing how the utility components are attached to the information bus 28

Figure 3.2 Two components engaging in the communication. 32

Figure 3.3 Showing a template and the tuple which are matching. 33

Figure 3.4 CACIP Class Diagram...36

Figure 4.1 Activity diagram for the RLS...43

Figure 4.2 A use case diagram of the Restaurant Locator Service. 44

Figure 4.3 Class Diagram showing how the Restaurant Locator Service object Relates To other objects of the CACIP model. 45

Figure 4.4 RLS prototype architecture as dictated by CACIP. 47

Figure 4.5 MainUI...48

Figure 4.6 New user registration UI...48

Figure 4.7 User authentication UI....................50
Figure 4.8 First choice prompt....................50
Figure 4.9 Second choice prompt. 50
Figure 4.10 Processing user's request....................51
Figure 4.11 Restaurant found, details are provided 51
Figure 4.12 Message throughput: Impact of the number of tuple messages....53
Figure 4.13 Round-trip time: effect of the number of tuple messages..............54
Figure 4.14 Role of Context-Awareness in mobile environments....................55

List of Tables

Table 3.1 Description of Component Bus Pattern....25

Table 3.2 Description of the component glue pattern....26

Table 3.3 Description of CACIP....27

Table 3.4 Overview design of CACIP architecture 34

Table 3.5 Comparing CACIP with some of existing schemes....40

Abstract

This research work focuses on the development of the Context-Aware Component Interfacing Pattern, CACIP, aimed at contributing to current developments in distributed communication and collaboration in mobile wireless environment. CACIP derives from the component bus and the component glue patterns as standard solutions to robust producer-consumer interaction in a mobile wireless environment. Communication between the participating components is managed by a common data structure referred to as the information bus. To showcase usefulness of the pattern, a prototype of an example service was implemented which was subsequently used as a testbed when evaluating the performance of the proposed scheme. We evaluated performance of the model in terms of message throughput, round-trip time and we finally measured the significance of context-awareness to the model. For message throughput, the model displayed a normal behaviour of any distributed system since as the number of components participating in communication activities were increased, the overall message throughput dropped. When measuring the model's round-trip time, i.e. the time taken to send a request message and receive back a response, we observed that increasing the number of messages exchanged among the participating components caused the model's round-trip time to increase. Again, this was an expected result. An experiment aimed at measuring the significance of context-awareness to the model was also conducted. The obtained results showed that when context data was not utilised, the number of inaccurate responses was higher compared to that of accurate ones. On the other hand, incorporating context-awareness produced the opposite of the first result, thus proving that context-awareness is a useful feature that mobile applications must adopt to improve their quality of service.

CHAPTER ONE

INTRODUCTION

1.1 Introductory Background

The technological advances in mobile wireless networks and widespread popularity of mobile devices have paved the way for the construction of new mobile applications. However, developing applications for mobile wireless environments comes with challenges peculiar to such environments. For example, in order for these applications to live up to the expectations of their users, they must be able to adapt to changes in context, and this requires that mechanisms be put in place to facilitate acquisition and utilisation of context information by mobile applications. Context is defined by Dey and Abowd as "any information that can be used to characterise the situation of an entity, where an entity can be a person, place or any physical or computational object" [1]. As Couderc and Kermarrec stated in [2], context may include information such as user's preferences, platform capabilities, current location of a user, etc. Any software system that is able to "sense" changes in context and able to use context information to dynamically adapt services it provides accordingly, is said to be context-aware [1].

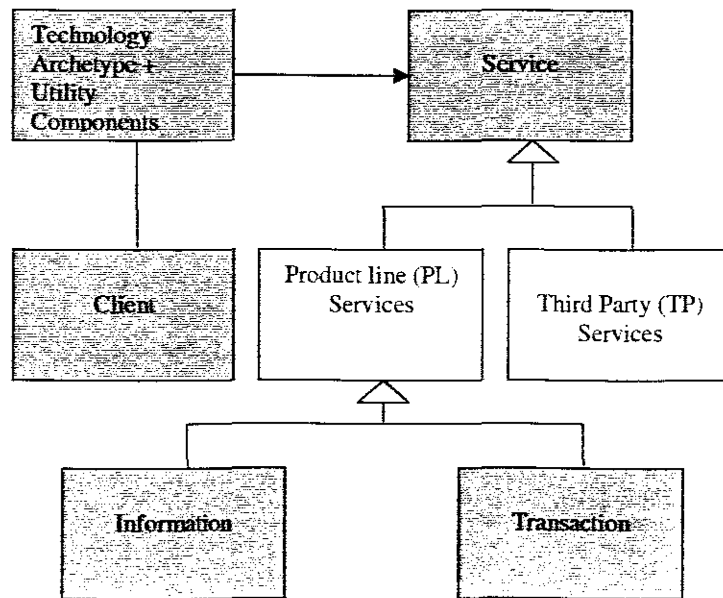

Figure 1.1 An m-Commerce Product Line reference Architecture [4].

With promising advances in context sensing and modelling systems [3], it is evident that context information will be an integral part and key component in mobile environments. Therefore, in order for mobile applications of the future to give nothing less than acceptable quality of service, they will have to incorporate some notion of context-awareness.

In line with the requirements mentioned above, an emerging product line m-commerce reference architecture shown in Figure 1.1 has been proposed. This architecture features five archetypes, where an archetype is a term defined by Bosch in [5] as "the core abstractions based on which a software system is structured." Each of them (*Client, Technology, Service, Transaction and Information*) is briefly described as follows:

i) *Client* – an archetype that abstracts any mobile device that can be used to access services provided by any product crafted from the reference architecture;

ii) *Technology* – a service provisioning environment through which context information will be served to other architectural components, such as Transaction and Information services;

iii) *Service* – a generic archetype for both *information* and *transaction* archetypes;

iv) *Transaction* – any service involving exchange of money and products between a supplier and a buyer, but excluding web content and

v) *Information* – content-oriented services such as downloadable music, pictures, etc.

The technology archetype was conceptualised as an infrastructure that defines how context data is captured, managed and then made available to other architectural components whenever they need it. The implication of this is that this archetype needs a standard mechanism to facilitate and support the serving of context information to other elements of the architecture. The only existing scheme with this kind of characteristic is Information Requirement Elicitation or IRE [6]. IRE is defined as the interactive communication protocol used by services to help users specify their service requirements with adaptive choice prompts. This means that users can be quickly guided to services they need with less effort. IRE was, therefore, adopted as the protocol for driving the *Technology* abstraction in the reference architecture [4].

The IRE-defined context was then restructured into specific utility services providers such as *data mining component, supplier and services data* component, *user profile and preferences,* as well as the *context sensor.* These components will actually be responsible for capturing context data and serving them to other components as required [4].

Guided by other previous research activities in the domain of context-aware mobile computing [1], these utility components were, in effect, defined to support, among others, the following context-aware features:

a) Presentation of context information to the user – this can be achieved, for example, when a mobile device is equipped with a GPS receiver (global positioning system) as one of context sensors that can present to the user his current location; and
b) Automatic adaptation of service behaviours offered by a mobile application according to the discovered context.

Figure 1.2 is the class diagram version of the product line reference architecture depicted by Figure 1.1. This class diagram shows, in terms of the UML terminology, how the entities that make up the reference architecture relate to one another. For example, Figure 1.2 shows that the **Technology** class is composed of the four classes representing the utility components – the context data sources.

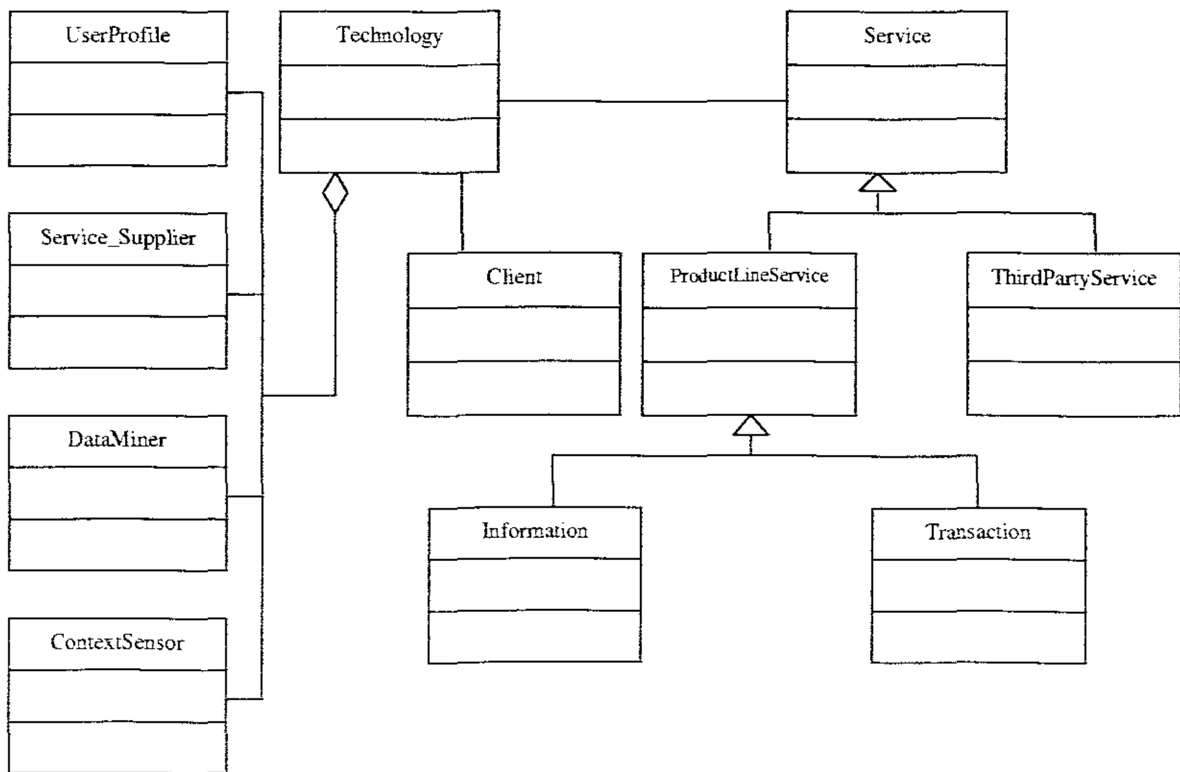

Figure 1.2 Class Diagram for the Mobile Commerce Product Line Reference Architecture.

1.2 Statement of the Problem

The reference architecture in figure 1.1 consists of clients and various application services which are expected to interact with the utility components, making requests for context information needed to configure the services they provide. In distributed environment such as the one in which mobile commerce operates, the components may be residing in different nodes in the network. Candidate answers are, therefore, needed to the following three research questions. First, which communication mechanism are different architectural components going to

use to acquire context data from the utility components? Second, how are the utility components themselves going to coordinate their activities when making context data available to other components? Third, in which common data format will context data be communicated to various architectural components? These issues need to be investigated, given the fact that the widely used standards for distributed systems such as remote procedure call (RPC) and its object-oriented descendants like CORBA are not well suited for wireless mobile environments. Though efforts such as mobile wireless CORBA [7, 8, 9, 10] have been made; however, wireless CORBA has not yet been proven to be a successful solution as it does not cater for context-awareness, which is a crucial feature that mobile applications of the future should have, to successfully deliver services that can be dynamically configured as changes in context occur. Besides, CORBA's main component, the ORB, can introduce considerable overhead on a resource limited mobile setting [9]. This research work is an attempt to investigate and come up with a context-aware model that addresses the research questions raised.

1.3 Research Goal and Objectives

Current mobile commerce systems do not have a context-aware protocol for delivering services to the clients; hence, the requirement for services to interface with some context databases, so that when the user makes a particular request, the system is able to respond according to the current context of the user.

1.3.1 Research Goal

The goal of this research is to come up with a suitable pattern for interfacing context-requesting mobile applications with context-providing components to achieve dynamic context-aware application.

1.3.2 Research Objectives

In order to realise the goal above, the following objectives were identified, which are to:

i) define how context data is exchanged among components possibly residing in different nodes of a distributed system;

ii) define a common data format in which context data will be served among different architectural elements, a requester and a provider;

iii) implement a prototype service to show the usefulness of the proposed component interfacing model; and

iv) evaluate performance of the model using the prototype as a test bed.

1.4 Research Methodology

The methodology adopted in this research is standard and it began with the investigation of existing inter-component communication models with a view to

understanding usage context, suitability for purpose, and interface styles. Then a model construction phase followed when knowledge gathered from the literature became the foundation for the model proposed here. Specifically, the model proposed in this work was composed from two well-known component interfacing patterns, namely the component glue and component bus patterns. To support context-awareness four utility components were defined guided by the IRE work as described in [6]. The result of this activity was a new composite context-aware component interfacing pattern. Next, the design and implementation of a prototype using a restaurant locator service as an example were done to show the usefulness of the new pattern. Finally, the model was evaluated for performance using the implemented prototype as a testbed.

1.5 Organisation of the Dissertation

The remainder of this dissertation is organised as follows: Chapter two is a review of relevant inter-component communication models, found in the literature, as the basis of the component interfacing approach proposed in this dissertation for context-awareness. In chapter three, CACIP is presented as a Linda-like Information Bus oriented architecture that is used by client services to obtain context data. Chapter four consists of two parts: firstly, the implementation of the Restaurant Locator Service as a prototype for proving CACIP, and secondly the evaluation of how CACIP performs with respect to scalability, Quality of Service, and Context-Awareness. Finally, Chapter five presents the conclusion which

consists of how the research objectives were achieved, the limitations of the results obtained and possible future work for further research and results.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

Developing mobile commerce applications requires that developers consider many requirements peculiar to the wireless and mobile setting. These requirements range from those that concern the mobile users themselves to those that are dictated by the environment in which these applications are to run. Some of these requirements are overviewed in [11]. In the mobile wireless environment users are always mobile and services are distributed over different nodes in the network. Users access these services using their mobile handheld devices such as mobile phones, PDA's, etc. It is quite common that as user's context changes, his/her service requirements will also change; as a result a mobile application should dynamically reconfigure itself to reflect these changes. This phenomenon is known as context-awareness. This is to say that mobile commerce applications should be aware of the changes in context of both the user and the environment in which they execute, otherwise the services they offer will not satisfy the current needs of the user. To achieve context-awareness, mobile distributed applications should define mechanisms for capturing and managing context data so that it is made available to all services in the system that want to use it at any time.

To address this requirement, the mobile commerce reference architecture [4] introduced in chapter one was proposed, which defined context data sources as parts of a 'utility' abstraction. Each of these data sources is possibly hosted by a different node in the network, and as such is expected to provide context data like user profile, user preferences, and so on, to the system components so that the services provided are personalised and configured accordingly. This research is to contribute to the current challenge of finding the best possible inter-component communication pattern that can be employed in a mobile wireless environment to facilitate the interaction between context data producers and context data consumers. This challenge has been partially addressed in various computing contexts and an array of inter-component communication models have been proposed as a result. In the following section we review some of these models.

2.2 Inter-component Communication Models

This section reviews some of the existing schemes based on the following characteristics:

a) Component interfacing/communication in a service-based environment – focuses on the way a model being reviewed achieves inter-component communication.

b) Context-Awareness for reconfiguration – researches on context-awareness [12, 13, 14, 15] have shown that in order to improve QoS

for mobile applications, context-awareness is an important feature that these applications must incorporate. Particular attention is paid to see how some of the existing models make provision for context-awareness.

2.2.1 Dynamic Adjustment of Component InterActions (DACIA)

DACIA [16] is a modular architecture that allows applications to be built through the composition of software components implementing individual operations or functions. In DACIA model a component is referred to as a PROC (Processing and Routing Component). 'A PROC can apply some transformations to one or multiple input data streams. It can synchronise input data streams; it can split the items in an input data stream and send them alternately to multiple destinations. PROCs represent the basic building blocks for an application. They can be interconnected in multiple ways, according to certain rules and restrictions. PROCs are identified system-wide using a unique identifier obtained by combining the host ID where the PROC originated and a counter maintained by the host' [16]. PROCs communicate by exchanging messages through input and output ports. DACIA model supports both asynchronous and synchronous communication between PROCs. For synchronous communication, the PROCs must be located on the same host and, to reduce overheads, the thread that executes the action associated with the source PROC will also execute the message handling routine of the connected PROC. In the asynchronous case,

the messages received by a PROC are inserted into the PROC's message queue. Every PROC has a thread that handles the messages in the queue, usually in FIFO order.

The following primitives are used to establish connection and disconnection between PROCs and to allow them to engage in communication activities:

- *connectProcs (procID1, portNol, procID2, portNo2)* – connect two PROCs. The PROCs can be local or remote;
- *DisconnectProes (procID, int portNo)* – disconnect two PROCs;
- *output (portNo, message, isSynchronous)* - send a message to the specified output port, either synchronously or asynchronously and
- *input (portNo, message, isSynchronous)* – receive a message on the specified input port.

Although DACIA provides a somewhat clearly-defined model of communication between entities; however it does not provide support for context-awareness which is feature that is needed by mobile applications of the future.

2.2.2 Scalable Timed Events And Mobility (STEAM)

The inter-component communication presented by STEAM [17] is based on events. Components that make up a mobile application interact using an event-based communication in order to exchange information. STEAM defines two types of components, namely, those that provide others with data (producers),

and those that make use of that data (consumers). So, consumer components subscribe to particular event types, so that when an event occurs all subscribing components will be notified accordingly. This kind of setup achieves decoupling between the communicating components because consumer components only subscribe to events rather than to specific producer components. However, like DACIA, STEAM does not provide support for context-awareness.

2.2.3 Java Event-based Distributed Infrastructure (JEDI)

JEDI [18] is another framework that uses events to achieve communication among components, is based on the notion of A*ctive Objects (AOs)*. An AO is an autonomous computational unit that performs an application-specific task. Each AO interacts with other AOs by explicitly producing and consuming events, where events are a particular type of message. Events in JEDI do not include any information about their recipients, thus achieving decoupling between the active objects. Figure 2.1 illustrates a logical architecture of JEDI. An event is generated by an AO and sent to a component called the event dispatcher (ED).

The ED notifies the event to those AOs that have explicitly declared their interest in receiving it (event recipients). An AO declares the classes of events it is interested in by invoking an event subscription operation. It also can stop accepting events of a given class by invoking the *unsubscribe* operation. Event subscription and unsubscription can be invoked at any time during the AO lifetime.

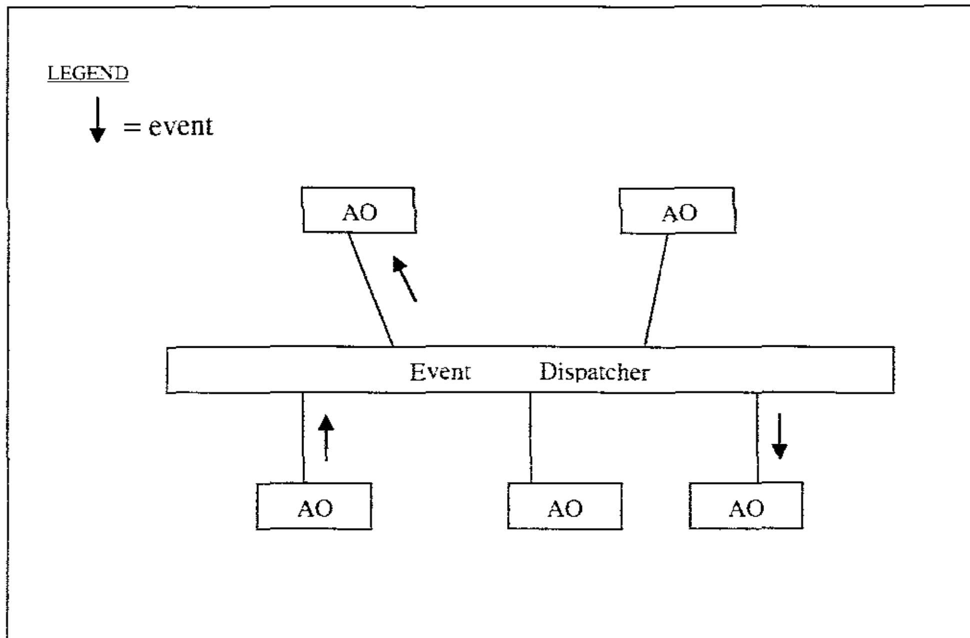

Figure 2.1. A logical view of JEDI [18]

The notification of events is accomplished asynchronously with respect to their generation. This model looks like it has something for each of the two characteristics of interest to this research, and for this reason, its structural organisation was borrowed as the basis for components arrangement in the proposed model.

Figure 2.2 is the UML class diagram version of the JEDI architecture (Figure 3.1), showing how the objects of this architecture are related to one another.

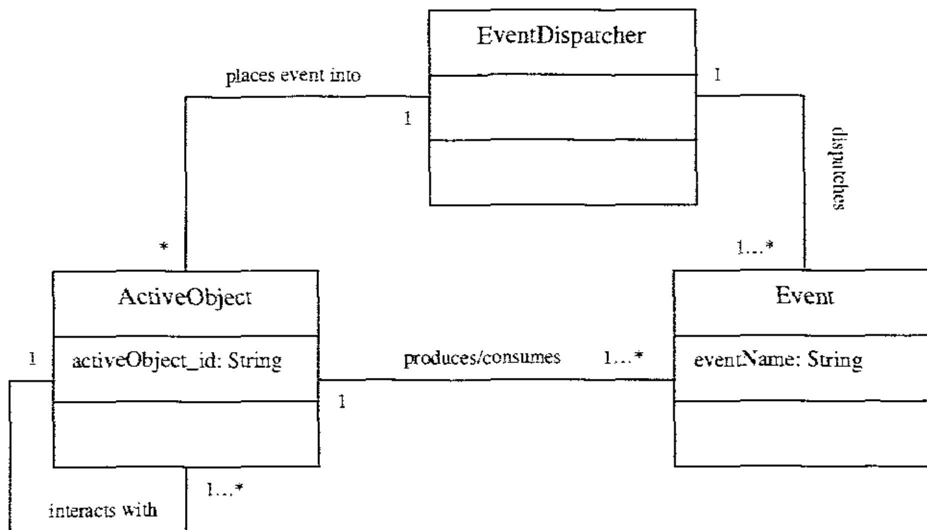

Figure 2.2. Class Diagram for the JEDI Architecture.

2.2.4 Gaia: Model for Developing Context-Aware Applications

The Gaia [19] model allows components or agents of a mobile application to acquire contextual information easily, reason about it using different logics and then adapt themselves to changing contexts. The model achieves this by providing a context infrastructure depicted in Figure 2.3. Agents that are involved in the Context Infrastructure within Gaia include *context providers, context synthesiser, context consumers, context provider lookup service, context history service* and the *ontology server.*

a) *Context Providers.*

Context Providers are sensors or other data sources of context information. They allow other components (or Context Consumers) to query them for context information. Some Context Providers also have an event channel where they

keep sending context events. Thus, other components can either query a Provider or listen on the event channel to get context information.

b) Context Synthesisers.

Context Synthesisers get sensed contexts from various Context Providers, deduce higher-level or abstract contexts from these simple sensed contexts and then provide these deduced contexts to other components.

c) Context Consumers.

These are context-aware applications that get different types of contexts from context providers or context synthesisers. They then reason about the current context and adapt the way they behave according to the current context.

d) Context Provider Lookup Service.

Context providers advertise the context they provide with the *context provider Lookup Service*. This service allows components to find appropriate *context providers*.

e) Context History Service.

Past contexts are logged in a database. The Context History Service allows other agents to query for past contexts. There is one such service in a single computing environment

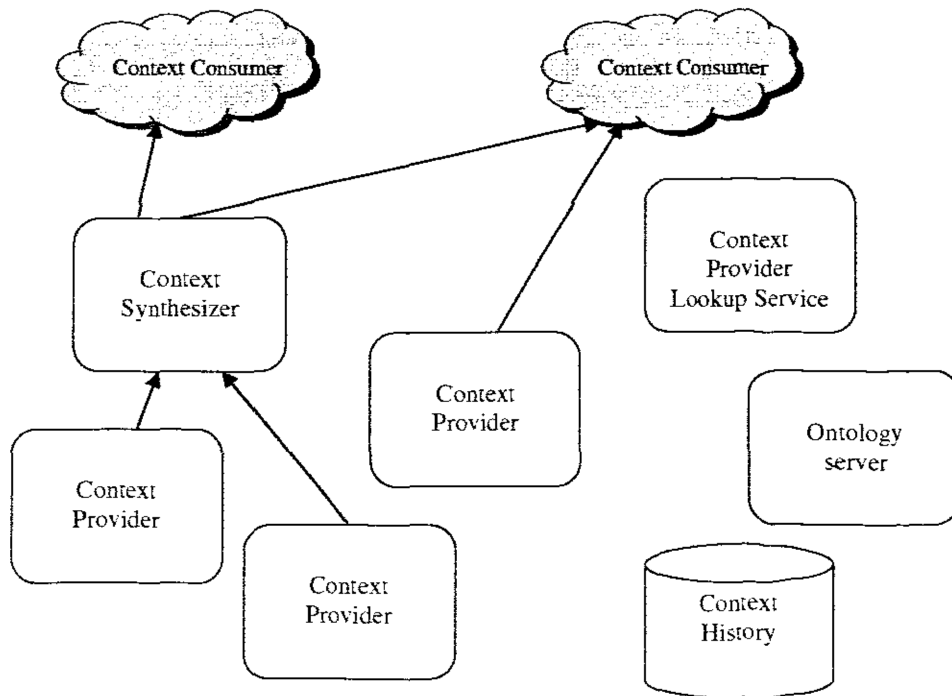

Figure 2.3 Gaia Context Infrastructures [19].

f) Ontology Server.

The *Ontology Server* maintains ontologies that describe different types of contextual information. There is one *Ontology Server* per ubiquitous computing environment.

All these components form part of the Gaia infrastructure to support context-awareness. Gaia uses CORBA to enable distributed components to find and communicate with one another. This type of communication partially address the two characteristics of interest, the only concern is the fact that it uses CORBA which has not yet been proven to be a successful solution for inter-component communication in the mobile wireless setting.

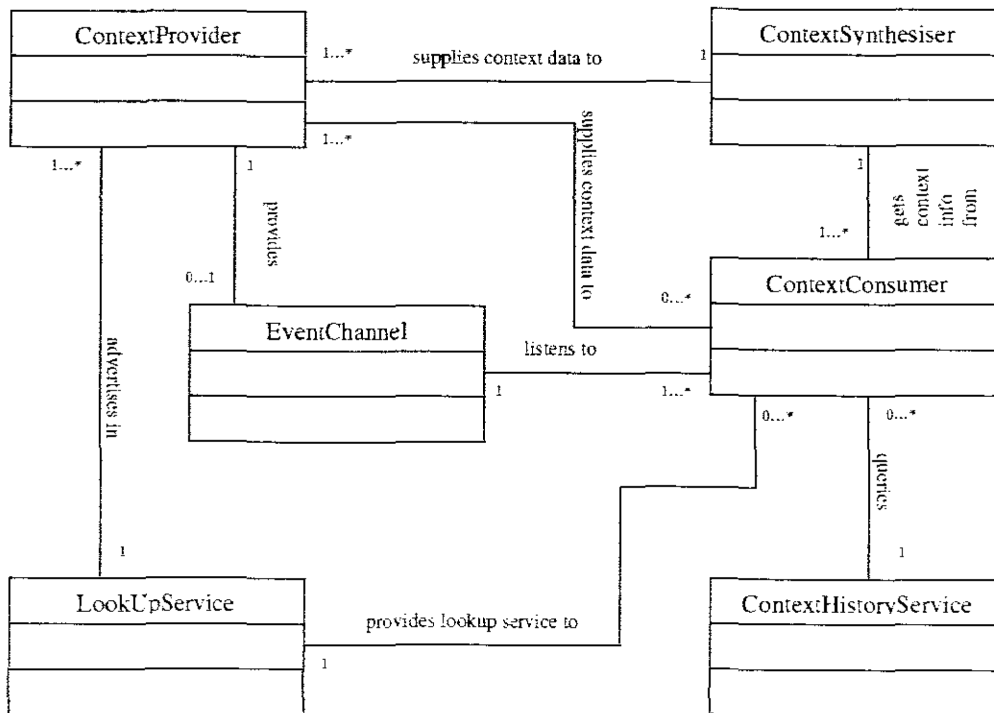

Figure 2.4 Class Diagram for the Gaia Context Infrastructure.

Figure 2.4 depicts the UML class diagram version of the Gaia architecture illustrated by Figure 2.3.

2.2.5 The Hydrogen Context-Framework

The Hydrogen Context-Framework [20] is a context aware framework aimed at supporting the development of context-aware mobile applications. It comprises three layers to separate the concerns of interacting with the physical sensors, storing and maintaining the context from the application itself. These layers are the *Adapter Layer, the Management Layer,* and *the Application Layer.* The *Adapter layer* deals with the low-level raw context data collected from different

context sensors. The *Management Layer* is responsible for providing and retrieving context data and shares the data with client devices using peer-to-peer communication. The *Management Layer* achieves its function through its core component called the *ContextServer*. All applications which use context data provided by the *ContextServer* form part of the *Application Layer*. Applications communicate with the *ContextServer* using either XML-streams or serialised Java objects. One major flaw of this framework is that it places all three layers in one device. This raises serious concerns because it means that all processing of context data takes place within a mobile device; and mobile devices are often resource limited.

2.3 The Proposed Solution Approach

In this dissertation a Context-Aware Component Interfacing Pattern (CACIP) is proposed as a model that is well suited for use by mobile applications. The proposed model was crafted from the component bus and component glue interaction patterns [21], and borrowed JEDI's [20] structural organisation for the arrangement of different components. We then introduced context-awareness by defining the utility components – the context data sources responsible for capturing, managing and making sure that the context information is made available to all other architectural elements or services which depend on this information to dynamically configure themselves accordingly. The component bus interaction pattern [21] adapted Linda's [22] **out** and **rd** operations for

interacting with the bus by defining the *writeTuple* () and *readTuple*() versions of these operations respectively. In chapter three a detailed account of the proposed model will be given to show how the two patterns were combined into a new compound component interfacing pattern.

2.4 Software Design Patterns

Software design patterns are core abstractions extracted from successful recurring problem solutions in software design. The idea behind patterns is to support software re-use by allowing software designers re-use solutions that have been tested and proven to be successful and standard solutions in a particular problem domain. Patterns also help in documenting architectural design decisions, and facilitate communication between stakeholders through a common vocabulary [23]. A set of interlinked patterns for a specific domain is known as a *pattern language* [24]. The history of pattern languages started in the field of architecture [25] and patterns have been adopted and applied successfully in different computing domains including, but not limited to, User Interface Design [26, 27, 28, 29] and Web Design [30, 31]. Agerbo and Cornils summarise the benefits of design patterns as follows [32]:

1. They encapsulate experience.

2. They provide a common vocabulary for computer scientists across domain barriers.

3. They enhance the documentation of software designs.

Design patterns can be generally classified into three main categories, namely Creational patterns, Structural patterns, and Behavioral patterns [33]. Creational patterns provide solutions for class or object instantiation. Structural patterns are design patterns that represent the relationships between entities. Behavioral patterns provide solutions for interactions between entities. A lot of work has been published on design patterns [34, 35, 36], and a common view that can be deduced from these works is that patterns represent the collective wisdom and experience of the software development community, so that the application of the relevant patterns should lead to higher quality systems with predictable behaviour. For example, in [37] the authors presented a pattern-driven analysis and design method called PAD. In this work the authors argue that patterns do not prove to be useful only during design, but can also be applied successfully throughout the different phases of software development.

It is without a doubt that the usefulness and the benefits of design patterns are being supported by the large community in the field of computer science. In this work we also opted to address our research problem by using the idea of software design patterns as a standard solution.

CHAPTER THREE

THE PROPOSED MODEL

3.1 Introduction

As highlighted in chapter one, the main goal of this research was to come up with a context-aware component interfacing strategy that would enable mobile applications to dynamically configure the services they provide according to the current needs of the mobile user. This dynamic configuration of services is a requirement often dictated by the change in context of both the execution environment as well as that of the mobile user him\herself. An acceptable interfacing strategy would be expected to define how context data was to be made available to different services provided by the system whenever needed, and how the involved components would interact to actually achieve this. In this chapter a detailed description of the proposed model is presented.

The model addresses two challenges namely, distributed communication and collaboration, and producer-consumer interaction. The component bus and the component glue interfacing patterns [21] were chosen as standard solutions to the challenges. The result of this exercise was the Context-Aware Component Interfacing Pattern (CACIP). The new pattern is documented in the rest of the chapter starting with its architecture and how it serves the purpose of facilitating

interactions between the utility components. Next, the information bus is described as the facilitator of asynchronous communication among architectural components. A common message format is introduced for collaboration and interaction among components. A set of operations is then defined upon this data structure to support communication. Finally, the chapter is wrapped up with a specific design of the solution pattern called CACIP.

3.2. The Context-Aware Component Interfacing Pattern (CACIP)

A pattern is a reusable solution whose context can be matched with new problems provided there is a fit. Schmidt et al. define a pattern as a recurring solution to a standard problem [38]. A pattern can, therefore, be documented by specifying a *context* in which it is used, the *problem* it tries to solve and the *solution* to the problem. As examples, the component bus and the component glue patterns [21] are illustrated in Tables 3.1 and 3.2 respectively while CACIP is illustrated in Table 3.3.The CACIP model is a compound derivative from these two component interaction patterns. The component bus interaction pattern defines a way of binding the communicating components without their being explicitly dependent on others. This pattern actually tries to address a situation where components that make up a distributed system are required to communicate and

Table 3.1. Description of Component Bus Pattern.

Context	You are to build a system composed from components that will be distributed across a network. These components are expected to collaborate to get the job done; and therefore an interaction mechanism has to be put in place to facilitate inter-component communication.
Problem	"Systems consisting of components with many interdependencies can behave unpredictably or fail to operate altogether if explicit bindings they depend upon aren't established properly or connections are lost."
Solution	"Bind components to an information bus that manages the routing of information between communicating components to remove explicit dependencies from the components themselves. Define interaction protocols that not only specify interfaces required for components to participate, but also the nature of interactions occurring between them."

collaborate to get the job done. Coordinating the process of instantiating and binding these components together could be a very challenging task especially when there are many explicit dependencies among them. To solve this problem,

Table 3. 2. Description of the component glue pattern.

Context	Components that are to be assembled to form a new system can have some incompatibilities in terms of their interfaces due to perhaps the fact that they were acquired from different developers. Yet these components have to work together as a single coherent system as one component may need services provided by another component in order for it to accomplish its functions.
Problem	"Assembling components that are incompatible in the way they communicate or interact can be time consuming and inevitably complicates system architecture."
Solution	"Create a glue code to act as an adapter for incompatible components, or as a mediator between peers. Only build full-fledged components when glue doesn't meet all of your requirements."

component bus interaction pattern introduces the idea of an "information bus" - a persistent data structure that manages the routing of information between the components participating in an interaction, that is, from those that produce information to those that consume it. This routing of information is managed

Table 3. 3 Description of CACIP.

Context	You are to build a mobile distributed system whose components will be expected to communicate and collaborate to get the job done. These components may be built in-house or acquired from third-party sources; and therefore an appropriate interfacing mechanism for these components has to be designed.
Problem	Developing distributed systems for mobile wireless environments can be a challenging exercise, especially as the contextual situations of both users and execution environment are always changing. As a consequence, services provided by different components of the system have to be aware of these changes so that they can adapt accordingly.
Solution	Use component bus interfacing pattern [21] to ensure asynchronous communication among participating components. Through its persistent data storage (information bus), the component bus interfacing pattern provides the mechanism to deal with the problem of frequent network disconnections. To support context-awareness, define mechanism for acquiring and managing contextual information (e.g. utility components in the case of CACIP) which will ensure that context data is always available to other components of the system anytime they need it. Use component glue pattern in the case of third-party components displaying incompatibilities.

dynamically as the participating components can attach to and detach from the bus without affecting the integrity of others. Component glue interfacing pattern,

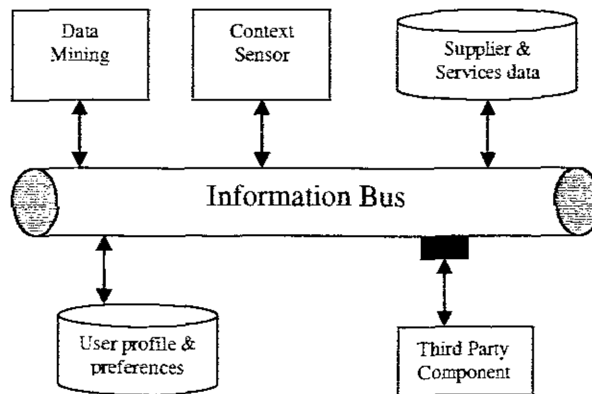

Figure 3.1 The Context-Aware Component Interfacing Pattern – showing how the utility components are attached to the information bus.

on the other hand, helps tie incompatible components together without increasing coupling, and it prevents designers from having to introduce protocols riddled with adapters into the architecture. The proposed model (CACIP) leverages concepts from these two component interaction patterns.

The heart of the CACIP architecture is the information bus introduced to coordinate inter-component communication as illustrated in Figure 3.1. Therefore, there is no direct component-to-component communication, but rather, inter-component communication is managed by the information bus. All participating components are required to register with the information bus, specifying types of messages they are interested in. This registration will ensure that every time a new message is written into the information bus, a component that is interested in it is notified. An interested component can then read this message from the bus and, if necessary, a response message can be created and written back into

the bus. The format of CACIP messages is discussed in subsection 3.2.2. Next, a discussion is given on the information bus and why this kind of component interaction set up is desirable in the mobile environment.

3.2.1 The Information Bus

Traditional client/server inter-component interaction mechanism is not suitable in the mobile wireless environment; rather an asynchronous model of communication need to be put in place which can enable inter-component interaction in a mobile distributed environment. CACIP adopts the idea of an information bus as a persistent data structure that facilitates this kind of communication between the context-producing and context-consuming components. Rather than interacting directly, components just place their messages (tuples) into the information bus, and in turn, the intended recipients retrieve these messages from the bus whenever they want to. The information bus is implemented and administered across a network in one or more servers. Several participating components on the same or different nodes can access the bus simultaneously. Some will be adding tuples to the bus and others removing them.

3.2.2 A Common Message Format

One of the objectives of this work as mentioned earlier is to define a common data format to be used for exchanging messages. It is important for all entities

that participate in interactions to adopt a common message format, thus making it much easier for them to process messages. The proposed model defines a message format in terms of a tuple. A tuple consists of a set of fields. A typical tuple will look something like the following:

[$Field_1$, $Field_2$, $Field_3$..., $Field_n$]

Each field of the tuple can be designated to hold data for specific purpose. For example, the tuple: ***["curr loc: Richards Bay", "usr id: smzuma"]***, consists of only two fields of type *String*. This tuple could be one of many tuples generated by the utility components as it contains the current location (context data) of the user identified as *smzuma*.

3.2.3 Operations for Interaction

The component bus interaction pattern [21] stems from Linda [22], the first programming language that popularised the idea of inter-process communication via a shared data space or tuplespace. Inter-process communication in Linda is accomplished by using the following simple primitives, namely: *out*, *in*, and *rd*. The ***out*** primitive inserts a message into the dataspace. When a process wants to read a tuple from the space, it invokes the ***in*** primitive which blocks this process until the tuple is found. Upon finding the tuple, the process removes it from the dataspace. The ***rd*** primitive functions in the same manner as the **in** primitive except that instead of removing the tuple from the dataspace the processes only use a copy of it.

The Linda operations overviewed above provide functionality closest to the requirements of our model and therefore, inspired us to design the following operations for use by CACIP components during their interaction activities.

- *attach ()*

This operation allows components to attach to the information bus. This will make the bus aware of all components which are ready to engage in communication.

- *registerEvent()*

The participating components use this operation to register for events they would like to be notified of. This ensures that every time a tuple is written into the information bus, the relevant component is notified. During event registration, a component specifies the type of tuples it expects from other components. A component achieves this by defining a *template* tuple. Subsection 3.2.4 gives more details about templates.

- *detach ()*

The participating component uses this operation to terminate the interaction.

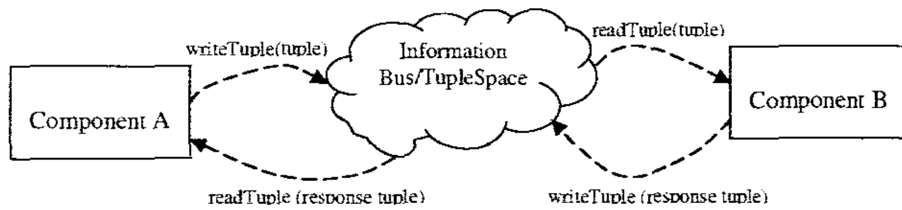

Figure 3.2. Two components engaging in communication.

- *createTuple()*

To create a new tuple, a component will invoke this operation.

- *writeTuple()*

This is for inserting a newly created tuple into the information bus. When the *write* event occurs, the bus notifies all components registered for this event, that is, the consumer components.

- *readTuple()*

The consumer components invoke this operation to read tuples of their interest from the bus. Only tuples that match a component's template can be read from the bus.

- *deleteTuple()*

The participating components can be able to delete tuples from the information bus using this operation.

Figure 3.2 shows how components can use some of these operations to interact with one another via the information bus.

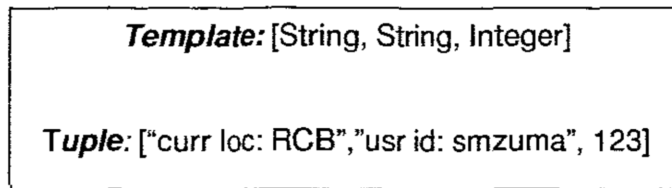

Figure 3.3 Showing a template and the tuple which are matching.

3.2.4 Retrieving Tuples from the Component Bus

When a new tuple is written into the information bus, how does the system figure out which component registered to receive that particular tuple? When a component registers for an event, it also is required to define a *template tuple* which specifies the format of the tuples it expects from other components (producer components). Thus, everytime a *writeTuple()* event occurs in the information bus - which effectively inserts a new tuple into the bus - all the components (consumer components) that registered for such an event are notified. Upon receiving this notification, each consumer component checks if its *template* matches this newly written tuple. If the two match, then the component can retrieve the tuple from the bus using the *readTuple()* operation. In order for the *template* and the tuple to match, they must have equal number of fields and the corresponding fields must be of the same type. As shown in Figure 3.3, the template does not necessarily have to provide values for its fields, but each field

Table 3.4 Overview design of CACIP architecture

BEHAVIOUR	OBJECTS
Inter-component communication in CACIP is coordinated by a persistent data buffer referred to as the information bus. All participating components (utility components, third party components and product line components) use a common message/ data format in the form of a tuple. Before engaging in any communication all the components need to attach to the information bus, and must register with it so that it notifies them of any events (insertions of new tuples into the bus) occurring in the information bus. Such event registrations will ensure that whenever a new tuple is written into the bus, relevant components are notified and these components can then read the tuple from the bus.	**infoBus: InformationBus** **utilComp: UtilityComponent** **tuple: Tuple** **mService: MobileService** **event: Event**
ALGORITHM	**OPERATIONS**

//Create information bus object
 	
InformationBus infoBus = new InformationBus()
 	
Until system shuts down **do**
 	
// initialise events including registration
 	
DataMining mining = new DataMining()
List.add(mining).registerEvent(*eventName, templateTuple*)
 	
UserProfileAndPref usrProf = new UserProfileAndPref()
List.add(usrProf).registerEvent(*eventName , templateTuple*)
 	
ContextSensor cntx = new ContextSensor()
List.add(cntx).registerEvent(*eventName. templateTuple*)
 	
ServiceSupplierData servSuppl = new ServiceSupplierData()
List.add(servSuppl).registerEvent(*eventName. templateTuple*)
 	
MobileService mService = new MobileService()
List.add(mService).registerEvent(*eventName. templateTuple*)
 	
// create an attach thread
While attachQueue not empty **do**
InfoBus.attach() // the parameter of this attach
// operation can be any component
// object e.g. a utility
// component or a mobile service.
attachQueue.next()
EndWhile
 	
// create an event occurrence thread
While List not empty **do**
event = List.next()
event.registerEvent()
EndWhile
EndUntil
	createTuple()
	readTuple()
	writeTuple()
	deleteTuple()
	registerEvent()
	attach()
	detach()
	updateContextInfo()
	supplyContextInfo()
	makeRequest()

must be given a type. Figure 3.3 illustrates a scenario in which a template and a tuple match because they have the same number of fields and their fields are of the same type.

The overview of the object model of CACIP is summarised in Table 3.4 in terms of its behaviour, the involved objects, algorithm as well as the operations that the objects use during their interaction. The **behaviour** quadrant gives a narration of how inter-component communication is achieved according to the proposed model, CACIP, while the **objects** section of the Table lists the objects that may be involved during interaction. These are instances of the classes in Figure 3.4. The **algorithm** quadrant shows how components that participate in the interaction attach to the bus and register for events. In this algorithm, the *infoBus* object created. All the participating component objects i.e. *mining, usrProf, cntx, servSuppl and mService*, are required to attach to this object in order for them to be able to exchange their messages/tuples. This is handled by the 'attach' thread of the algorithm. The last column lists the methods or operations that are implemented by the objects.

A class diagram is useful for illustrating entities in a model and their relationships. Figure 3.4 is a class diagram of CACIP in the context of the m-commerce reference architecture introduced in chapter one. Each class is described in the following:

1. **UtilityComponent:** this class represents any of the four components identified as sources of context data to be used by services provided by an m-commerce application;

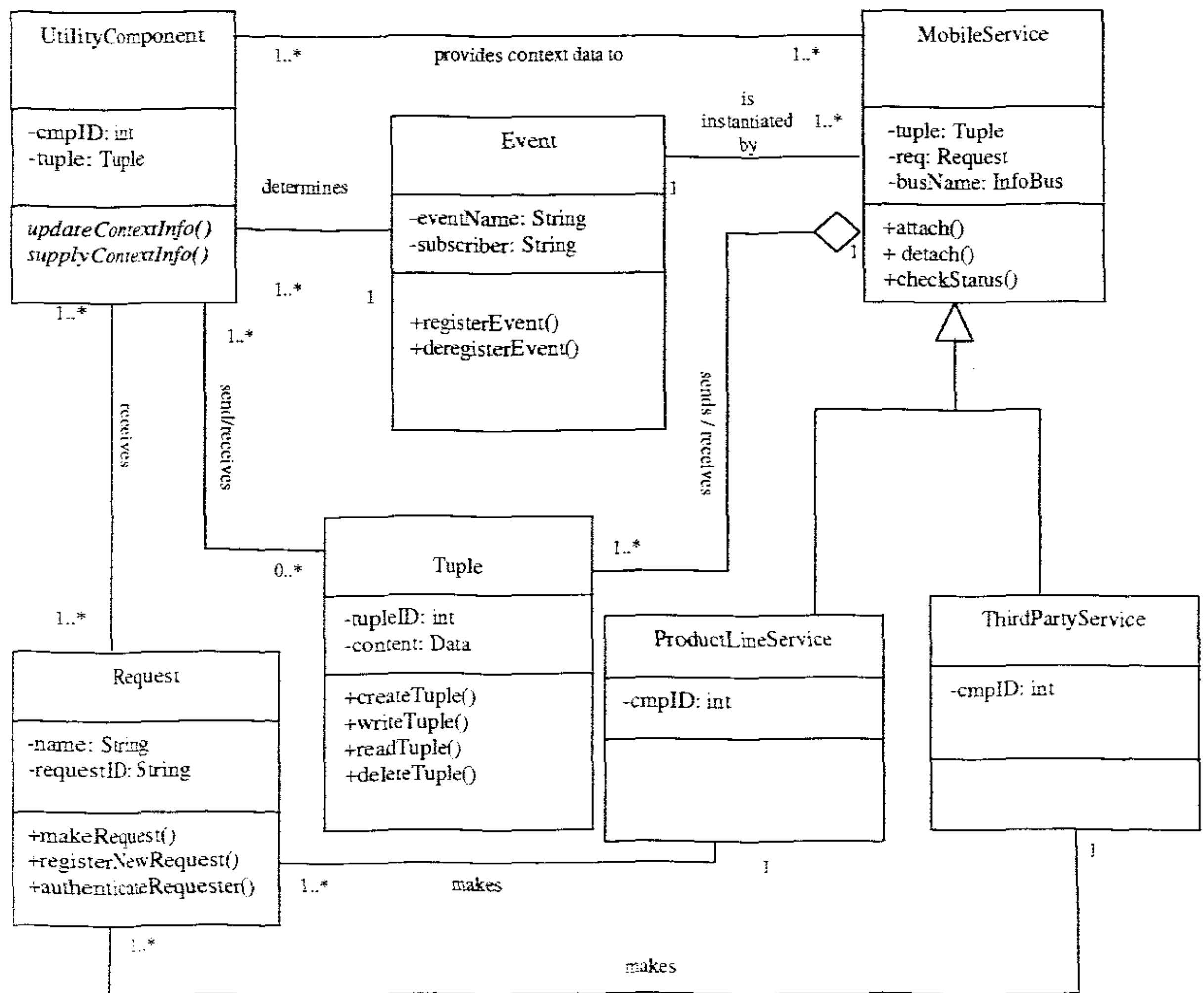

Figure 3.4 Class Diagram for CACIP in the context of the m-Commerce Reference Architecture.

2. **MobileService:** this is a super class to the ProductLineComponent class and the ThirdPartyComponent class;
3. **Event:** this class defines valid service requests that components can make and needs to be instantiated by these components before issuing any request for service;

4. **Request:** this class represents a request for service provided by utility components as facilitated by CACIP;

5. **Tuple:** this class represents the format in which the messages are sent during inter-component communication. Every component that wants to engage in any communication with others has to send its messages in the form of tuples;

6. **ProductLineComponent:** represents all services that will be provided by all products composed from our departmental m-commerce reference architecture introduced in chapter one and the

7. **ThirdPartyComponent:** this class represents services that are not part of the product line, but rather, are loaded from external sources such as games from the Internet.

In chapter four the prototype design and implementation of an example service will be presented as proof for usefulness of the CACIP model.

3.2.5 Context-Awareness in CACIP

"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" [39]. To cater for context-awareness, CACIP model was, therefore, augmented with the utility components which provide database services that store context data such as user profile and preferences, sensor data, service description and supplier data etc. In mobile environments context-awareness is a crucial feature that helps mobile applications to dynamically configure the services they provide as

dictated by both the change in context of the environment in which these services are to execute, as well as that of the mobile user himself. When developing mobile applications, it is, therefore, necessary to take into account such issues as how context data are going to be made available to different services that the system provides, considering the fact that these data are stored and managed by components distributed over the network. This requires that a good inter-component communication model suitable for mobile environment is devised.

3.3 Dynamic Configuration Provisioning with CACIP

In CACIP the utility components provide context information such as user's current location and user preferences. In this section we give a description in terms of an example scenario as to how dynamic configuration of services can be achieved with CACIP using such context information. We use a scenario of a restaurant Locator service (RLS) whose prototype implementation details are given in chapter four.

In this scenario, Peter is a tourist roaming around a city and having a PDA installed with RLS which uses CACIP as a support for context-awareness. Peter is a registered user of the system and, therefore, his profiles are known to the system. Peter wants to visit one of the restaurants in the city, and he then logs on to the system so that he can make a request for service. After authentication, RLS immediately queries the utility components for Peter's current location and preferences. With this information RLS can now start its search for a restaurant

based on the contextual relevancy to Peter, for example the type of food that Peter usually enjoys when visiting a restaurant (Peter's food preference). Peter's

current location and preferences, therefore, help RLS to narrow down its search for relevant restaurants instead of searching the whole database of restaurants available in the system. This ensures that in the end RLS returns only a list of restaurants serving the type of food preferred by Peter, which are also in the vicinity of his current location. Now, next time Peter travels to another city and plans to visit a restaurant there, the system will be able to adjust itself according to Peter's new context.

3.4 Comparison of CACIP with Existing Schemes.

The model proposed in this work is a contribution to existing research results such as those listed in Table 3.5. CACIP as an idea in the making is not as well-developed as any of the systems listed here, but it proves that components in a mobile wireless operating environment can interface seamlessly, with constantly updated context and fault-tolerant distributed component interaction.

Table 3.5 Comparing CACIP with some of existing schemes

Model Name	Component Interfacing Style	Context-Awareness (Supported?)
DACIA: Dynamic Adjustment of Component Interaction [16]	Communication through multiple protocols. Synchronous communication	Not supported
STEAM[17]	Uses producer/consumer type of communication based on events. Consumer components are required to subscribe to particular events so that when an event occurs, all the subscribing components are notified accordingly.	Not supported
Gaia[19]	Inter-component communication is based on CORBA. The limitation is that CORBA has not yet been proven to be successful in mobile wireless settings.	Provides an infrastructure for supporting context-awareness.
Hydrogen Context-Framework [20]	Uses XML-streams or serialised Java objects for inter-component communication	Provides a 3-layer infrastructure for supporting context-awareness
CACIP: Our Proposed model	Asynchronous communication supported by the information bus which is responsible for routing of information to different components involved in the interaction.	Utility components provides mechanism for supporting context-awareness

CHAPTER FOUR

DESIGN AND IMPLEMENTATION OF THE PROTOTYPE

4.1 Introduction

In the previous chapter the proposed model for interfacing components in a mobile environment was presented. The model showed how components of a distributed mobile commerce application could be interfaced in order for them to work together to fulfil their task, and how context-awareness could be incorporated to assist in the delivery of services that are personalised and dynamically configured as dictated by current context of a mobile user. To show the usefulness of the proposed model, this chapter presents the design and the implementation of an example prototype service, that is, a restaurant locator service. Performance evaluation for the model and the results obtained are also presented.

4.2 Prototype Development for the Restaurant Locator Service (RLS)

The most important objective of implementing this prototype is to show how **MobileService** instance interacts with mobile environment services in a distributed application using the CACIP model, how data are exchanged among the components and to show how context awareness can improve service delivery in the mobile setting. Briefly, the flow of events in this service is as follows: the system authenticates a user, and tries to obtain the user's context-specific preferences and context information such as user's current location from the utility components. This information could be obtained by determining, for example, the type of food the user has recently been ordering whenever visiting a restaurant. The system then displays a choice prompt to the user, suggesting food and asks if the user is happy with this suggestion. The system is able to make this suggestion because it keeps track of user's service choice patterns. This information is managed and supplied by the utility components. The user can either accept system's option or suggest a different one. The system then contacts the supplier and services database to find a restaurant - within the current location of the user - that serves the type of food s/he has selected.
Finally, the system returns information such as the name of the restaurant that meets the search criteria, its physical address as well as the telephone number.

Figure 4.1 is a UML activity diagram showing actions taken during a session of locating a restaurant.

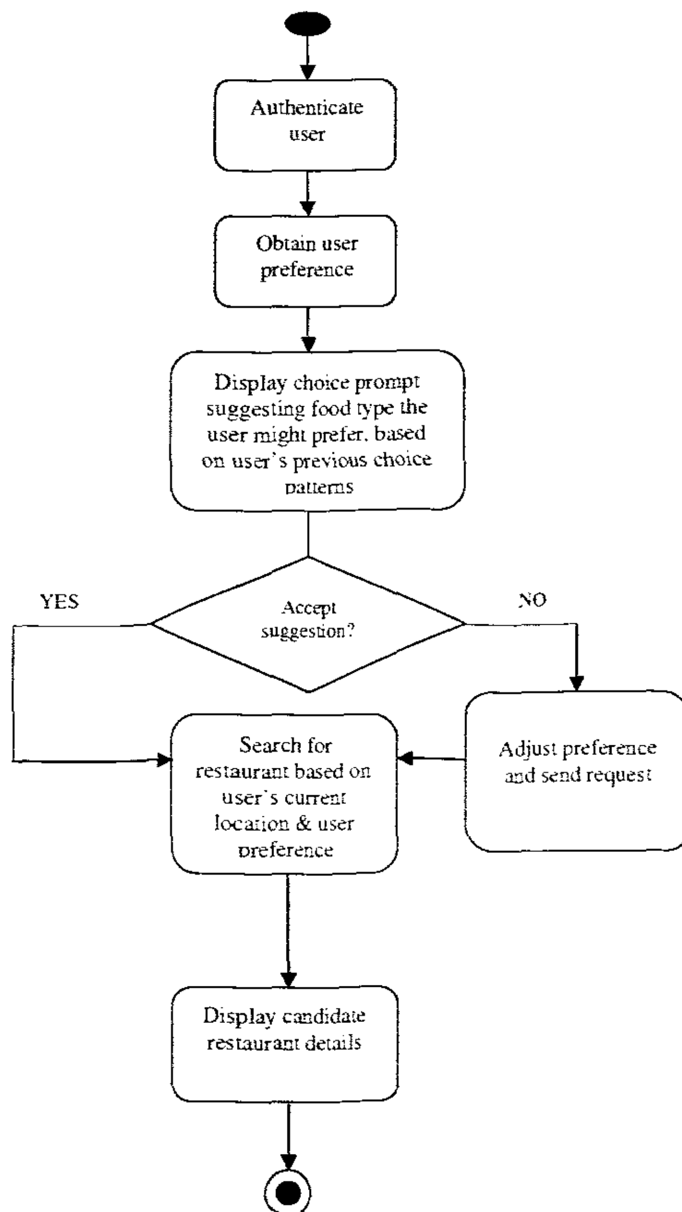

Figure 4.1 Activity diagram for the RLS.

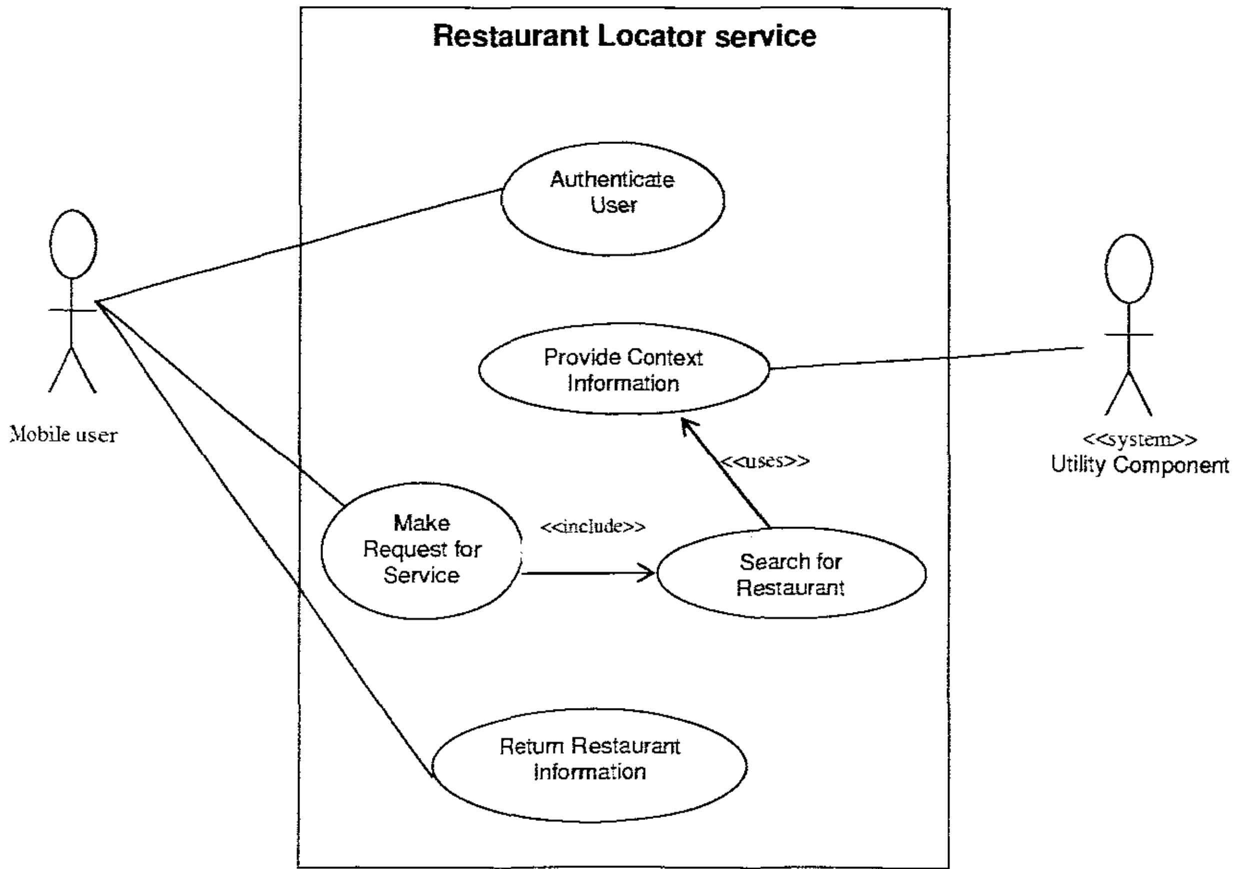

Figure 4.2 A use case diagram of the Restaurant Locator Service.

Figure 4.2 depicts a use case diagram of the Restaurant Locator Service showing actors involved in different use cases.

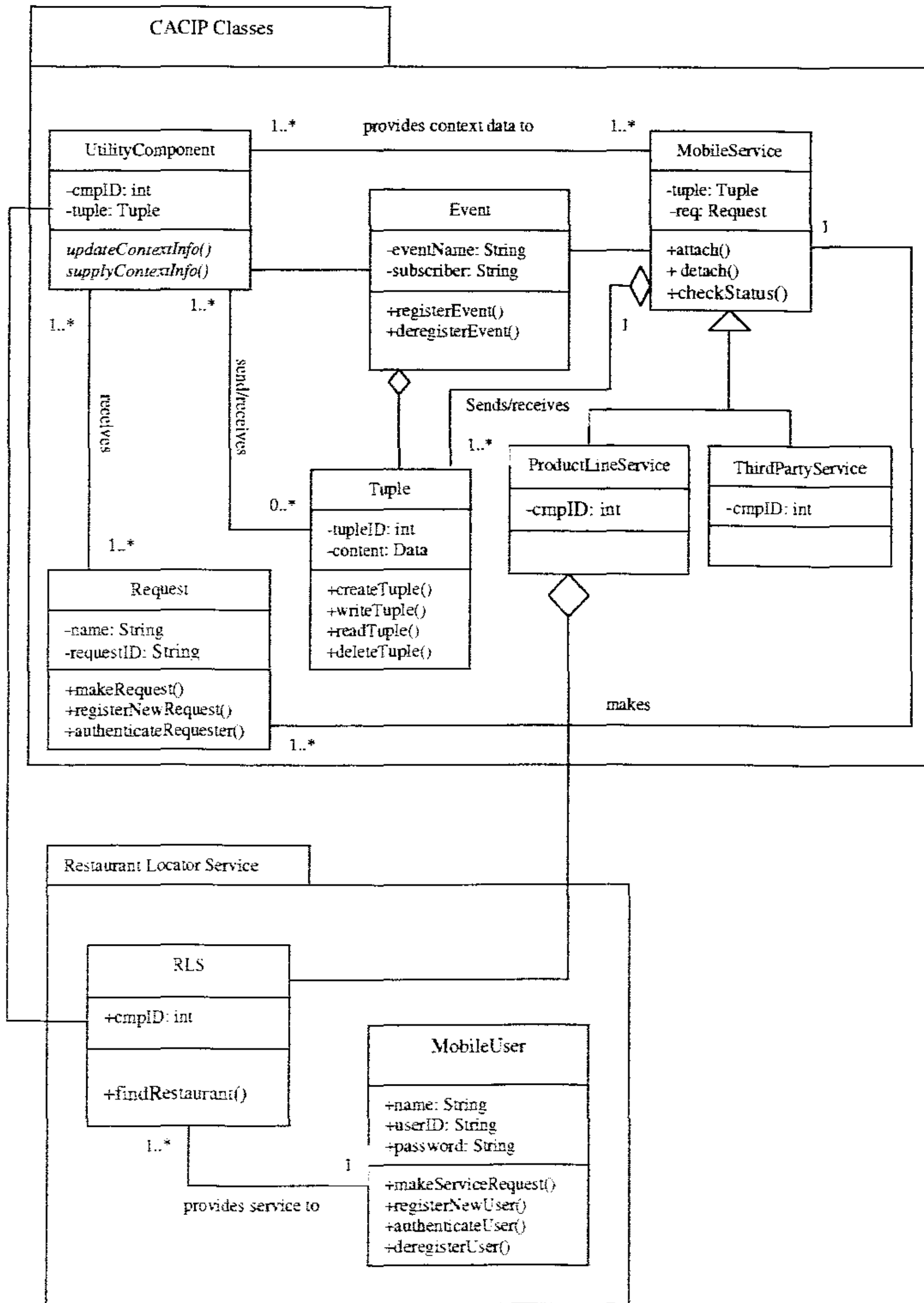

Figure 4.3 UML package diagram showing a relationship between the Restaurant Locator package and the CACIP package.

Figure 4.3 shows that the Restaurant Locator service package relates to the CACIP package via the ProductLineService interface. The UtilityComponent object interprets requests made by the MobileService object into events, tuples and so on and sends back the requested information to RLS via the ProductLineService interface. Figure 3.4 is about CACIP as a model, while Figure 4.3 shows how RLS makes requests for context data from CACIP's utility components.

4.3 Environment Specification

To realise the implementation of the Restaurant Locator prototype the following tools were used: JBuilder 5 – an Integrated Development Environment (IDE) for developing Java programs. We also needed to implement an Information Bus, which is the core component of the CACIP model as it is responsible for handling all messages exchanged among components engaging in a communication. IBM's TSpaces v.2.1.2 [40] provided us with a necessary API to implement the Information Bus component. There was a need to implement two databases, one for storing information about all restaurants available and for the other one was needed to store user profiles and preferences. We used MS Access for this purpose. MS Access was used simply because we were only implementing a prototype, otherwise database systems like MySQL – which store large amounts of data – could have been used instead. Our assumption was that the proposed

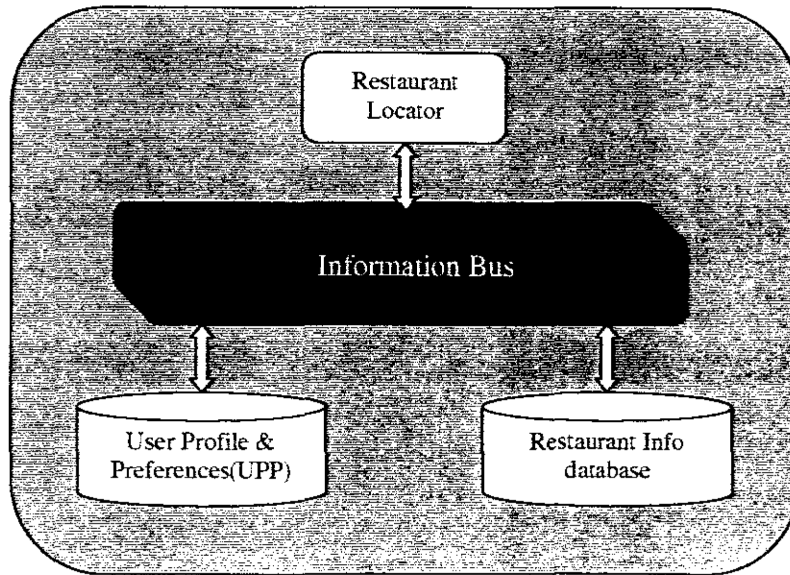

Figure 4.4 RLS prototype plugged into the CACIP architecture.

inter-component interfacing model is a distributed scheme, and therefore to emulate this we opted to use two machines, namely: a desktop PC equipped with a 504 MB RAM and a laptop with 512 MB RAM, both were Pentium 4's running Windows XP. The laptop was used in place of a mobile device to run a GUI through which a user was to interact with the restaurant locator prototype. All other components reside on the PC.

Figure 4.4 is a schematic diagram showing the Restaurant Locator Service (RLS) prototype plugged into the CACIP architecture. RLS depends on the context data supplied by the UPP component to configure its service to meet the user's current needs.

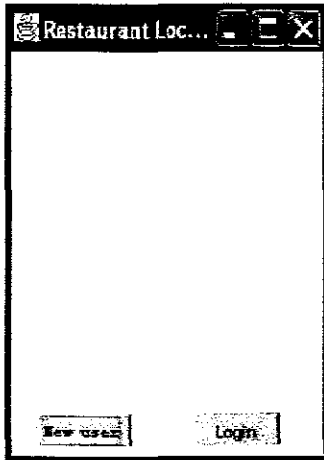

Figure 4.5 MainUI

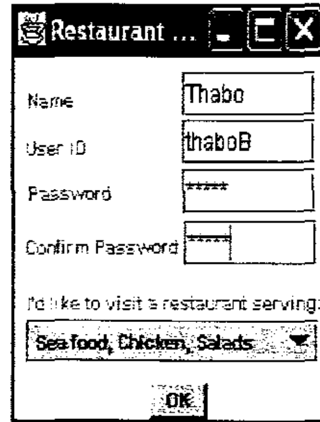

Figure 4.6 New user registration UI

4.3.1 RLS User Interface

In designing the graphical user interface of the prototype we had to make it as small as possible to emulate a mobile device such as a PDA or cell phone. We did not use any specific mobile device emulator because we were more concerned about prototyping the physical distribution of components in the mobile wireless environment. When the prototype is run, a user is presented with the main user interface shown in Figure 4.5. If it is a first time user, s/he needs to register with the system first before s/he can proceed to use the service. To register, a new user selects the ***New user*** button of Figure 4.5, which leads the user to the registration window as shown in Figure 4.6. Here, the user has to supply the service with his name and has to choose his/her user ID as well as a password. The user is also required to set his/her food preferences. Setting the

preferences will make it easier for the service to help the user when s/he returns next time. An already registered user, on the other hand, chooses the ***Login*** button in Figure 4.5 which takes him/her to the user authentication user interface depicted by Figure 4.7. Upon successful login or registration, the service displays a choice prompt – as shown in Figure 4.8 – where the type of food is suggested that the user might want to order when arriving at the restaurant. This information is supplied by the context data source, the UPP component to be specific. As shown in figure 4.8, the user can take the system's suggestion or choose a different option. The user is expected to also set his current location. Context data such as user's current location are supposed to be supplied by the *context sensor* component (see Figure 3.1, chapter 3). In the case of our prototype, the user's current location is used to narrow down the search for a restaurant to the area where the user currently is. After giving his/her location and confirmed the food preference, the user presses the ***YES*** button (Figure 4.8). This action effectively creates a new request which is sent to the information bus in the form of a tuple, and the message in Figure 4.10 is displayed. Upon receiving the request tuple, the bus notifies the *supplier and services data* component, which then retrieves the tuple and acts according to the message the tuple contains i.e. to search for a restaurant.

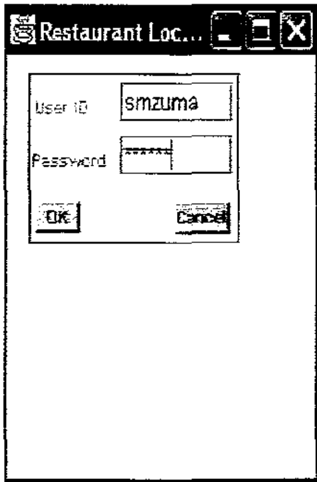

Figure 4.7 User authentication UI

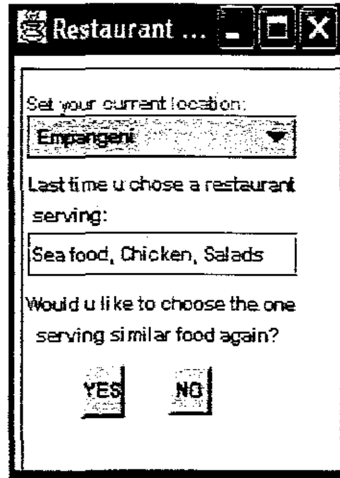

Figure 4.8 First choice prompt

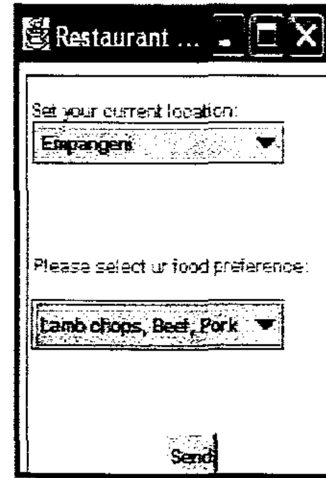

Figure 4.9 Second choice prompt

When a restaurant that meets the search criteria is found, the *supplier and services data* component creates a response tuple containing information such as the name of the restaurant, its physical address, telephone number or even the distance. The response tuple is written back into the information bus which is later retrieved and its contents are displayed to the user as shown in Figure 4.11. If the user chose the ***NO*** button in Figure 4.8, then the second choice prompt (Figure 4.9) is display for the user to set his/her food preference and current location. In a mobile environment the scenario we have just described might not go as smooth as expected since network disconnections are common characteristics of a wireless setting.

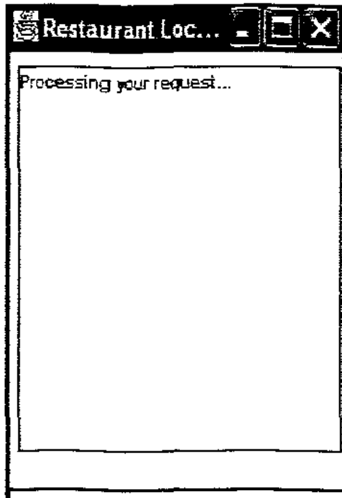

Figure 4.10 Processing user's request

Figure 4.11 Restaurant found, details are provided

The proposed model handles this problem by ensuring that the tuple is held in the information bus until the network connection is re-established and the bus can notify the intended component. This makes our model a much more reliable inter-component communication model. Next, we present performance evaluation for our proposed interfacing model using RLS as a testbed.

4.4 Performance Evaluation

In this section a description is presented of how performance evaluation for the CACIP model was carried out using RLS as our testbed. The results obtained are also presented. Three metrics were used to test the proposed model namely, system throughput, round-trip time and the impact of context-awareness on the

model. In order to test each metric, an experiment was designed which provided the mechanism for evaluating a specific aspect of performance. Each of these metrics is discussed in the next three subsections.

4.4.1 Message Throughput

The aim of this experiment was to measure the number of tuple messages the system was able to handle in a given unit of time. In conducting this experiment two parameters were varied namely: the number of participating components and the number of tuple messages each component was able to send in a fixed time period. This was achieved by constantly increasing the number of RLS component instances that were run simultaneously on different machines. This setup was done to emulate the idea of many clients requesting the same service simultaneously. The RLS component instances were sharing one instance of an Information bus and were creating and sending tuple messages to bus. As we varied the number of participating components and the number of tuple messages each of these components was sending, we observed how the system performed. Figure 4.12 illustrates a plot of the system throughput. As expected, the results show that the throughput is related to the number of participating components. The smaller the number of components engaging in communication, the higher the throughput peak point value. This is a normal behaviour of any distributed system.

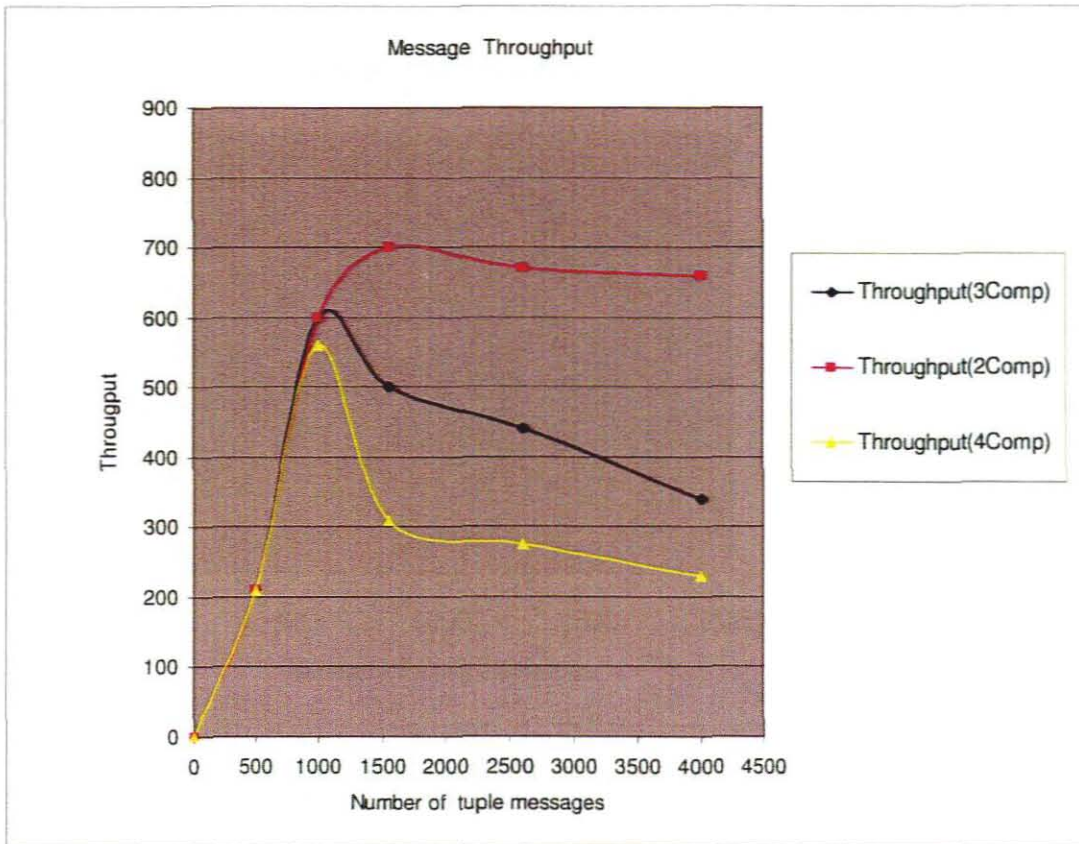

Figure 4.12 Message throughput: Impact of the number of tuple messages

4.4.2 Round-trip Time

The aim of this experiment was to measure the round-trip time of the system, i.e. time taken for sending a request or tuple message and receiving the results. To carry out this experiment two parameters were varied: the number of tuple messages and their size. The number of tuple messages was varied from 1 to 1500, while the tuple size was increased starting from 2KB up to 8KB. Figure 4.13 shows a plot of message round-trip time versus number of tuple messages.

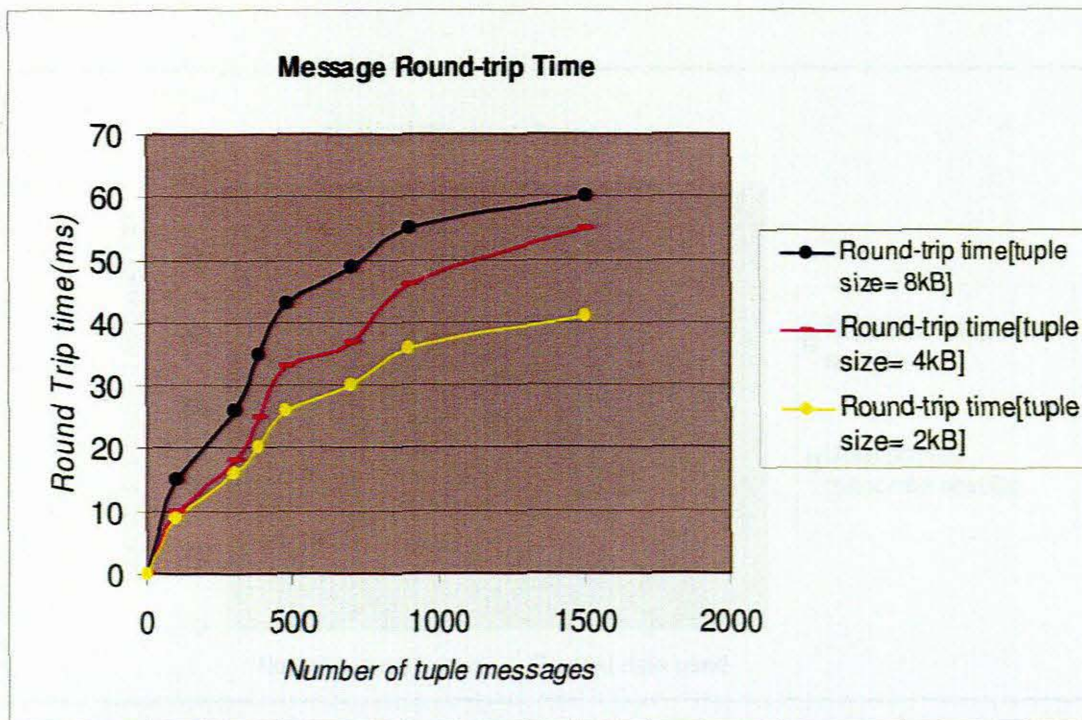

Figure 4.13 Round-trip time: effect of the number of tuple messages.

Each curve represents the round-trip time with a tuple message size equal to 2KB, 4KB, and 8KB respectively. This plot shows that as the number of tuple messages increases the overall round-trip time is increased. It was also observed that the overall round-trip time is proportional to the message size.

4.4.3 Role of Context-Awareness

Finally, we conducted an experiment that was aimed at determining the significance of context-awareness in the model. To achieve this, we measured the number of accurate responses versus inaccurate ones when:

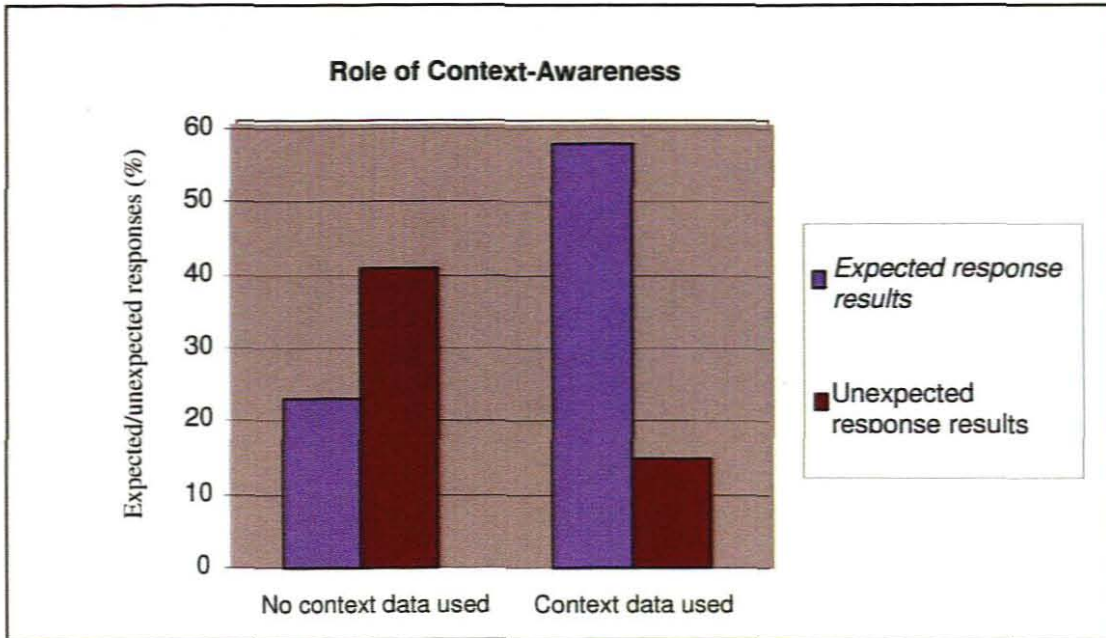

Figure 4.14 Role of Context-Awareness in mobile environments.

i) context data was used; and

ii) context data was not used.

Fifty requests for service were made in each of the above cases. In this experiment, the context data that used was user's location and preferences. The results obtained are illustrated by Figure 4.14. This plot shows that when context was not incorporated in the experiment the number of unexpected responses was higher compared to that of the expected results. On the other hand, when context data was utilised, the number of accurate responses was much higher.

This experiment serves as a proof that context-awareness is a useful feature that mobile applications must adopt to improve their quality of service.

CHAPTER FIVE

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

New mechanisms to support inter-component communication in mobile distributed systems are needed as the demand for the construction of these systems continuously grows due to technological advances in mobile wireless networks, and the increasing number of users of mobile devices. However, the construction of mobile applications is a challenging exercise as the environment in which these applications are to be executed is resource limited and is quite often susceptible to interruptions due to network disconnections. In order for mobile applications to meet the ever-changing service requirements of mobile users, they also need to incorporate context-awareness – which is one of very important features that mobile applications should possess to ensure that services they provide are easily adaptable when changes in context occur.

Following the proposal of mobile commerce reference architecture by our Department, it was, therefore, found necessary to embark on an investigation aimed at coming up with a best possible inter-component communication strategy that different components of the architecture were going to use to interact with one another. This communication model had to be designed taking

into consideration the characteristics and requirements of a mobile wireless environment as briefly overviewed earlier. In this research work, therefore, a Context-Aware Component interfacing Pattern (CACIP) was presented as our proposed solution approach. The model was derived from two component interfacing patterns namely, the component bus and the component glue interfacing patterns [22].

To achieve the goal of this research work the following objectives were set: first, to define how context data was going to be exchanged among different components in the architecture. This objective was achieved by specifying an asynchronous communication mechanism facilitated by the information bus component that was responsible for the routing of messages to their intended consumer components.

The second objective was to define a common message data format in which context data, or any data for that matter, was to be communicated to other architectural elements. To achieve this objective CACIP specified that all messages between components be communicated in the form of tuples.

The third objective of this research was to design and implement an example service to showcase usefulness of the proposed interfacing scheme. A prototype of a restaurant locator service was implemented. This service mainly used context information such as current location of a user and user preferences to fulfil its functions. Lastly, performance of the proposed scheme was evaluated using the implemented prototype as the testbed. During this process, we started

off by carrying out an experiment to measure the message throughput of the system, the result of which showed that throughput was related to the number of participating components. The smaller the number of participating components, the higher the throughput peak point value. The second experiment was aimed at measuring the time taken to send a requests or tuple message and receiving a response thereof. The result showed that the overall round-trip time increased as the number of tuple messages sent was increased. This kind of behaviour can, of course, be expected from any mobile distributed system.

Finally, an experiment aimed at measuring the impact of context-awareness on the model was conducted. The result of this experiment showed that incorporating context-awareness can significantly improve the quality of service as it was shown by a higher percentage of expected responses compared to unexpected ones when context data was used in this experiment.

5.2 Future Work

The results obtained from performance evaluation of the model presented in this dissertation helped in figuring out some areas that still need to be improved in order for the model to meet its full potential. When conducting an experiment to evaluate the model's throughput, the result showed that increasing the number of components participating in interaction activities somehow reduced the overall message throughput of the model. This fact pointed out the need for dealing with

this challenge to ensure that the system does not become degraded to the point where it is not functioning at all. Mechanism for handling the increasing number of users needs to be devised, and this is recommended for future work.

REFERENCES

[1] Dey A.K. & Abowd G.D. 1999. *Towards a Better understanding of Context-Awareness.* G.V.U Technical Report GIT-GVU-99-22. College of Computing, Georgia Institute of technology.

[2] Couderc P. & Kermarrec A. M. 1999. *Improving Level of Service for Mobile Users Using Context-Awareness.*18th IEEE Symposium on Reliable Distributed Systems, SRDS, p.24.

[3] DeVaul R., Sung M., Gips J., & Pentland A. 2003. *Mithril 2003: Applications and Architecture.* Proceedings of 7th IEEE Int. Symp. Wearable Computers, pp. 4-11.

[4] Adigun M. 2004. *Software Infrastructure for e-Commerce and e-Business.* Research Working paper RES-CSD-01. Centre for Mobile e-Services, University of Zululand.

[5] Bosch J. 2000. *Design and use of Software Architecture.* England: *Addison* Wesley.

[6] Sun J. 2003. *Information Requirement Elicitation In mobile Commerce.* Communications of the ACM, 46(12) pp. 45- 47.

[7] Lynch N. 1999. *Supporting Disconnected Operations in Mobile CORBA.* MSc. dissertation, Computer Science Department, University of Dublin.

[8] Adwankar S. 2001. *Mobile CORBA.* Proceedings of the 3rd IEEE Int. Symp. on Distributed-Objects and Applications. pp. 41 – 51.

[9] Haahr M., Cunningham R. & Cahill V. 1999. *Supporting CORBA Applications in a Mobile Environment.* Proceedings of the 5th annual ACM/IEEE Int. Conf. on Mobile Computing and Networking, ACM Press. pp36 - 47.

[10] OMG Telecom Domain Task Force. 1998. *White Paper on Wireless Access and Mobility in CORBA*, Version 2.

[11] Chung-wei Lee, Wen-Chen Hu & Jyh-haw Yeh. 2003. *A System Model for Mobile Commerce.* 23rd International Conference on Distributed Computing Systems Workshops (ICDSW'03), p. 364.

[12] Poladian V., Sousa J. P., Garlan D. & Shaw M. 2004. *Dynamic Configuration of Resource-Aware Services.* Proceedings of the 26th Int. Conf. on Software Engineering, pp604-613.

[13] Poladian V., Garlan D. & Shaw M. 2002. *Selection and Configuration in Mobile Environments: A Utility-Based Approach.* Position Paper in 4th International Workshop on Economics-Driven Software Engineering Research (EDSER-4).

[14] Cheng S., Huang A., Garlan D., Schmerl B. & Steenkiste P. 2004. *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure.* Computer, 37(10) pp46 -54.

[15] Garlan D., Poladian V., Schmerl B., & Sousa J.P. 2004. *Task-based Self-adaptation.* Proceedings of the ACM SIGSOFT Workshop on Self-Managing Systems (WOSS'04).

[16] Litiu R. & Prakash A. 2001. *DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications.* ACM SIGOPS Operating Systems Review, 2(35) pp31- 42.

[17] Meier R., Cahill V. 2002. *STEAM: event-based middleware for wireless ad hoc networks.* In 22nd international conference on distributed computing systems workshops (ICDCSW '02), pp639–644.

[18] Cugola G., Di Nitto E. & Fuggetta A. 2001. *The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS.* IEEE Transactions on Software Engineering. 9(27) pp827-850.

[19] Román M., Hess C., Cerqueira R., Ranganathan A., Campbell R. H. & Nahrstedt V. A *Middleware Platform for Active Spaces.* Mobile Computing and Communications Review. 6(4) p.65.

[20] Altmann J., Retschitzegger W., Schwinger W., Leonhartsberger G., Hofer T & Pichler M. 2003. *Context-Awareness on Mobile Devices - The Hydrogen Approach.* Proceedings of the 36th Annual Hawaii International Conference on System Sciences. pp292.

[21] Eskelin P. 1999. *Component Interaction Patterns.* Proceedings of 6th Conference on the Pattern Languages of Programming (PloP), USA.

[22] Gelernter D. 1985. *Generative Communication in Linda.* ACM Transactions on Programming Languages and Systems, 7(1) pp80 -112.

[23] Avgeriou P., Zdun U. 2005. *Architectural patterns revisited -- a pattern language. In 10th European Conference on Pattern Languages of Programs.*

ACM Transactions on Programming Languages and Systems, (EuroPlop 2005).

[24] Schmidt D., Coplein J.1995. *Pattern Language of Program Design*, volume 1 of Software Pattern Series. Addison Wesley.

[25] Alexander C. 1977. *A Pattern Language: Towns, Buildings, Construction*: Oxford University Press.

[26] Borchers J. 2001. *A Pattern Approach to Interaction Design*: John Wiley and Sons.

[27] Erickson T. The Interaction Design Patterns Page. http://www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html. Last accessed in October 2004.

[28] Tidwell J. 1999. *Common Ground: A Pattern Language for Human-Computer Interface Design.* http://www.mit.edu/~jtidwell/common_ground.html. Last accessed in October 2004.

[29] van Welie M., 2001. *Web and GUI Design Patterns.* http://www.welie.com/. Last accessed in October 2004.

[30] Graham I. 2003. *A Pattern Language for Web Usability.* Reading, MA: Addison-Wesley.

[31] van Duyne D., Landay J. and Hong J. 2002. *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. Reading, MA: Addison-Wesley.

[32] Agerbo E., Cornils, A. 1998. *How to preserve the benefits of design patterns.* Proceedings of the 13th Conf. on Object-Oriented Programming, Systems, Languages, and Applications. 33(10) pp134-143.

[33] Fowler M. 1997. Analysis *Patterns: Reusable Object Patterns*, Addison-Wesley.

[34] Roth J.2002. *Patterns of Mobile Interaction.* Personal and Ubiquitous Computing. 6(4) pp282-289.

[35] Ivory M., Megraw R. 2005. *Evolution of Web Site Design Patterns.* ACM Transactions on Information Systems, 23(4) pp463–497.

[36] Rossi G., Gordillo S., Lyardet F. 2005. A Pattern-Oriented Approach to Enhance Context Infrastructures. SAINT Workshops. pp170-173

[37] Hamza H., Chen Y.2005. *PAD: A Pattern-Driven Analysis and Design Method.* Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp132-133

[38] Schmidt D. C., Johnson R. E. & Fayad M. 1996. *Software Patterns.* Communications of the ACM, 39(10) pp37-39.

[39] Dey A.K. 2001. *Understanding and Using Context.* Personal and Ubiquitous Computing, 5(1) pp. 4–7.

[40] Available at IBM's website: *www.ibm.com.* Last accessed in August 2005.