MIDDLEWARE-BASED DEVELOPMENT OF CONTEXT-AWARE APPLICATIONS WITH REUSABLE COMPONENTS

Nearchos Paspallis

University of Cyprus, 2009

Driven by the proliferation of mobile and pervasive computing, there is a growing demand for context-aware, self-adaptive applications. Such applications benefit users by dynamically adjusting their offered services to the highly dynamic context which characterizes mobile and pervasive computing environments. To achieve this kind of sophistication, however, such applications must be capable of sensing the context, and autonomously reacting upon their knowledge on it. Inevitably, enabling this kind of behavior results in a measurable increase to the complexity of the underlying software.

Multiple approaches have been proposed for the development of context-aware, self-adaptive applications. Most of them focus on the actual capabilities and features of the resulting applications. However, streamlining the engineering of general context-aware, self-adapting applications is an important and challenging endeavor which has received less attention.

This thesis provides research contributions in the area of software engineering support for the development of context-aware, self-adaptive applications: It proposes development methods, supported by appropriate tools, aiming to facilitate the development and maintenance of context-aware applications targeting mobile and pervasive computing environments.

This is achieved in two basic dimensions. First, a novel development methodology is proposed allowing the design and implementation of context-aware, self-adaptive applications with

reusable components, called context plug-ins. This methodology is complemented with a model-driven development approach which automates the production of parts of the applications, extending reusability to sub-components. Second, a pluggable and modular middleware architecture is presented, facilitating the deployment and management of components comprising the context-aware, self-adaptive applications. At the same time, this architecture facilitates many recurring tasks pertaining context gathering, management and dissemination.

The results in this thesis have been evaluated both practically and theoretically. First, the proposed methodology and the underlying architecture implementation were evaluated in the context of two case-study applications. Second, the proposed middleware-based solution was compared to the state of the art over a number of requirements, as they were identified in the literature and summarized in this thesis. Finally, the development methodology was tested in the context of an undergraduate university course where the students were asked to develop context-aware applications as part of lab assignments. The students used the development approach proposed in this thesis, along with the accompanying middleware architecture, to build their own context-aware applications, partly by using the model-driven development approach also presented in this thesis. Their experience was documented and examined via a survey which justified that both the methodology and the middleware architecture add significant value in the hands of developers designing and implementing context-aware, self-adaptive applications.

The thesis concludes by summarizing its main contributions and by providing a discussion of key topics for future work.

MIDDLEWARE-BASED DEVELOPMENT OF CONTEXT-AWARE APPLICATIONS WITH REUSABLE COMPONENTS

Nearchos Paspallis

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

September, 2009

2009

APPROVAL PAGE

Doctor of Philosophy Dissertation

MIDDLEWARE-BASED DEVELOPMENT OF CONTEXT-AWARE APPLICATIONS WITH REUSABLE COMPONENTS

Presented by

Nearchos Paspallis

Research Supervisor _____
George A. Papadopoulos

Committee Member _____
Christian Becker

Committee Member _____
Marios D. Dikaiakos

Committee Member _____
Jeff Magee

Committee Member _____
George Samaras

University of Cyprus

September, 2009

to my loving wife, Fani

ACKNOWLEDGEMENTS

Many people have helped with the preparation of this thesis, both directly and indirectly. First, I would like to thank my advisor, Professor George A. Papadopoulos, who provided his support and guidance throughout the progress of my dissertation. Also, I would like to thank the members of my PhD thesis committee, Professors Christian Becker, Marios D. Dikaiakos, Jeff Magee and George Samaras, for their challenging but helpful questions and comments.

During the course of my thesis, I have collaborated and also have become good friends with many talented researchers from the IST-MADAM and IST-MUSIC projects. I feel privileged to have worked with these partners, from whom I have learned a lot. I would especially like to thank Roland, Michael, Mohammad and Kurt from the University of Kassel and also Frank and Romain from the University of Oslo, who shared their knowledge with me during these years.

Fani, Lakis, Konstantinos and Pyrros are boldly acknowledged as they have selflessly read my thesis and provided feedback and suggestions for improvement. Many thanks also go to my friends Aimilia, Despina, George, Konstantinos, Marios, Pericles, Pyrros and Vicky for making the lunch breaks in the last few years so entertaining and revitalizing.

My family has had an indirect but profound effect on my thesis. I am grateful for my parents, Loucas and Androula, for their persistent love and support. Both of them have been a great model for me since my childhood. My warmest thanks go also to my brother, Haris, for simply being the first and greatest teacher I ever had. Most importantly, I am indebted to my caring and loving wife, Fani, for her understanding and tireless support especially during the many weekends that were focused on this thesis.

Last, but not least, I would like to acknowledge the financial support I have received from the European Union during the course of my thesis. This funding came from the IST-MADAM and IST-MUSIC projects (Framework Programme 6, contract numbers 4169 and 35166 respectively).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Driven by the widespread adoption of mobile and pervasive computing devices, there is a growing demand for developing applications featuring context-aware and self-adaptive behavior. These features are needed to help applications adapt to dynamically changing mobile and pervasive computing environments, thus maximizing the utility provided to the end-users. This thesis discusses the software-engineering challenges which are inherent in developing context-aware, self-adaptive applications and proposes novel methods and tools which help developers create such applications more efficiently.

1.1 Motivation

With the widespread proliferation of mobile and pervasive computers, people increasingly use applications while away from their desktop, in effect realizing Weiser's vision of *ubiquitous computing* [139] [140]. Mobile phones in particular, have proved to be "*the most prolific consumer product ever invented*" [119], already counting 4.1 billion mobile subscribers—with the global population penetration exceeding 61%—as of early 2009 and with another billion expected to be added in the next few years [16]. Furthermore, embedded processors are also spreading quickly

to conquer the physical world: More and more modern machinery and appliances, coming out of production line, feature one or more embedded processors while in some products, like in modern automobiles, as many as 100 microprocessors are embedded, which is nearly as many as in the state-of-the-art Airbus 380 airliner [37].

These mobile and embedded microprocessor-based devices, however, differ from desktop computers in one fundamental way: their affinity to the physical world. As these devices are designed to assist humans in physical world scenarios, such as during driving, walking or traveling, they stand out in their ability to *sense* and *shape* the physical environment. For this purpose, special sensors and actuators are used to collect and process context information, and then apply the result of their reasoning accordingly. While many systems act on limited domains only, such as the *Anti-lock Breaking System* (ABS), this thesis is oriented towards generic programmable context-aware systems, capable of sensing and reacting to arbitrary types of context information.

The incorporation of sophisticated context-aware, self-adaptive behavior though, has some important implications on the development process as well. Most notably, the software required to realize this behavior typically implies significant increases to the complexity of the overall system. Of course, the increasing complexity of software is not a new problem. As early as in the late sixties, the term *software crisis* was used by software developers to describe what back then appeared as a bottleneck in productivity, caused by the increasing software complexity [96]. The same problem persisted throughout the eighties and the nineties, as it is vividly depicted by the exponential increase in software complexity observed in the U.S. space-flight program [59]. Naturally, as modern systems are still designed to be increasingly sophisticated, the problem of software complexity persists to nowadays. Technology evangelists even argue that [today] the problem of software complexity is bigger than ever and, as it is argued by Paul Horn in [74], "*dealing with it is the single most important challenge faced by the IT industry*".

This thesis argues that the added complexity which is required to implement the context-aware, self-adaptive behavior of mobile and pervasive applications indeed renders their development significantly harder and more challenging. This thesis presents a methodology complemented with a middleware architecture which together, it is argued, significantly simplify the task of developing, evolving, and maintaining applications featuring context-aware, self-adaptive behavior.

1.1.1 The vision of ubiquitous computing

Ubiquitous computing is often referred to as the post-desktop model of *Human Computer Interaction* (HCI). In this model, humans escape the stereotypical desktop-based interaction with computers (i.e., via keyboard-mouse-display), and rather interact with computers through everyday objects and activities.

Widely considered the father of ubiquitous computing, Mark Weiser described a vision of technologies that "[...] *weave themselves into the fabric of everyday life until they are indistinguishable from it*" [139]. This approach is based on the fundamental observation that the ratio of computers per person is constantly increasing. For instance, while this ratio was originally below the unit (mainframe era), it quickly reached the one-to-one value (personal computing era) and is now advancing to values well above the unit (ubiquitous computing era). As this ratio increases, so does the attention required by the users in order to maintain and interact with these devices and the applications deployed in them. This fact, actually, is already experienced today by many users who typically own a mobile phone, one or more laptop and desktop computers, portable music players, etc. This has prompted some specialists to claim that "*the most precious resource in a computer system is no longer its processor, memory, disk or network, but rather a resource that is not subject to Moore's law: User Attention*" [129].

From another point of view, it is argued that pervasive computing (a synonym of ubiquitous computing) is the evolution of mobile computing [122]. While mobile computing per se builds on the foundations of distributed systems, mobile networking and energy-aware systems [121], pervasive computing extends this with four additional research thrusts: *smart spaces*, *invisibility*, *localized scalability* and *uneven conditioning* [122]. The efficient use of smart-spaces refers to the idea of bringing the physical world together with the conceptual world (as that is modeled in computers). Invisibility refers to what Weiser described as "*seamless integration* [of computing] *to the fabric of our everyday life*". The idea of localized scalability refers to a method for avoiding intensive interaction with the user, by employing methods which decrease the affinity of users with services based on the distance between them. Finally, uneven conditioning refers to the need for a transition period (and space), where traditional computing applications and infrastructure are gradually replaced by others, introducing the features of pervasive computing.

Although the term of ubiquitous computing was coined many years ago, some confusion still exists regarding whether its vision has been already achieved or not. Even today most ubiquitous computing-related papers reference Weiser's vision as a fundamental basis for their research. However, many of these works still refer to the realization of ubiquitous computing to be chronically placed in the "*proximate future*" despite that this vision was formed almost two decades ago [26]. On the other hand, others argue that "*the age of ubiquitous computing is here* [presently]*: a computing without computers, where information processing has diffused into everyday life, and virtually disappeared from view*" [61].

Regardless of how exactly is ubiquitous computing defined and to which extend it has been already realized, it still remains one of the most interesting topics of research enabling a highly promising market. While ubiquitous computing may never reach its full potential, there are still many exciting applications that can be developed to improve the quality of our lives. To enable

the creation of such applications however, we need better development support comprising of conceptual models, methods and tools. These should aim at mitigating the complexity which is inherent in understanding and developing sophisticated context-aware, self-adaptive applications.

1.1.2 The role of context-awareness

The term of context awareness was first studied [137] and defined [125] just a few years after Weiser originally documented his vision of ubiquitous computing. At that point, context was defined as "*the location of use, nearby people, hosts and accessible devices as well as changes to these things over time*". As context-awareness was primarily aiming to overcome the limitations of mobile computing, it was often viewed as merely location-awareness. For instance, the first context-aware applications were predominantly location-aware, such as the Active Badge system [137], the ParcTab system [138], and the CyberGuide [86].

As more researchers got engaged in the study of context-awareness though, its definition evolved [31]. One of the most commonly referenced works defines context as "*any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves*" [44, 45].

Evidently, this definition is interaction-oriented: it conceptualizes the properties characterizing a user interacting with a computer application or service, and the environment in which the interaction takes place, including the user and the computing infrastructure themselves. As it will be discussed later on, the software engineering approach described in this thesis is aligned with this definition of context.

In relation to ubiquitous computing, context-awareness is considered one of the main enabling technologies. For instance, ubiquitous computing envisions a future where the interaction with

the users is minimized and raised to a subconscious level, whenever possible. However, in order for computing systems to decide on behalf of the users, they need to keep track of as much of the contextual information as possible: available resources, possible means of *User Interfacing* (UI), the user preferences and profile and past user choices, the user state and the physical environment.

In [66], it is argued that although ubiquitous computing has become a hot research topic, context-awareness remains in its infancy with very few examples of successful context-aware applications. Henricksen claimed that this situation was mainly due to the inherent software engineering challenges associated with: the analysis and specification of context-aware applications; the acquisition of context data from sensors, and subsequent management and dissemination; and, the design of suitable context-aware applications.

Today, although more context-aware applications are developed and deployed, the same software engineering challenges continue to hinder widespread adoption of context-awareness in everyday computing. To enable context-aware applications, and thus help realize the vision of ubiquitous computing, we need to identify and tackle the software engineering challenges that prevent their development.

1.2 Software engineering challenges

Software engineering is a discipline that is concerned with all aspects of software production [128]. Just like any other form of engineering, it aims at providing better processes yielding higher cost efficiency. To achieve this, software engineering incorporates a systematic and organized approach guiding the developers to use appropriate tools and techniques depending on the problem to be solved, the development constraints and the available resources.

While the principles of software engineering can be applied to the production of general software systems, the development of context-aware, self-adaptive applications is characterized by

properties which elevate some of the engineering challenges. The following subsections summarize just a few of the most relevant challenges. An extensive discussion of relevant challenges and requirements, as they are identified in the literature, is presented later in this thesis, in the analysis of related work.

1.2.1 Software complexity

As mentioned earlier, it is argued that the most important challenge in engineering general context-aware systems is *software complexity*. In particular, context-aware behavior implies that the applications must be autonomous, capable of operating in dynamic computing environments and respond to ever-changing user requirements [66]. However, implementing this kind of sophisticated behavior can become quite an intimidating and complex task, especially in the context of heterogeneous and resource constrained devices. For instance, autonomous systems should employ adaptive decision-making algorithms, which take previous system experience into consideration and use it to improve their effectiveness. These algorithms must be aware of the environment in which they operate and capable of responding to changes that affect the utility provided to the users. The users are also important variables in this equation, as they also have varying requirements depending on their tasks at hand. It is further argued that in order to make context-awareness more useful, these algorithms should act proactively by anticipating the users' intentions [133] [142], thus rendering the underlying system even more complex.

1.2.2 Heterogeneity and interoperability

By definition, ubiquitous computing is about going beyond the desktop model to interfacing humans with the real, physical world. It is thus natural to assume that the developed software must be capable of accommodating numerous mobile and embedded device platforms. As most

of these devices are capable of communicating with each other, appropriate protocols and standards are required to facilitate interoperability. Most importantly, the context information must be abstracted with standardized and scalable models, enabling interoperability among various and heterogeneous devices, systems and applications.

1.2.3 Resource limitations

While evolving computing beyond the stereotypical desktop model has some obvious advantages, it also introduces some noteworthy constraints. For instance, the fact that computing moves closer to the users and their everyday activities results to an inevitable move away from continuous power supply. Also, because of the implied mobility, the importance of size limitation is further elevated. These factors result in more resource constraints, such as memory and storage size, which must be taken into account when building context-aware, self-adaptive systems.

1.2.4 Modularity

As a consequence of the two previous challenges, the importance of modular architectures and systems becomes more evident. Ubiquitous computing settings are highly heterogeneous, consisting of various devices, systems, protocols and applications. In such scenarios, it is unlikely that one size fits all. It is rather expected that the underlying software, which powers the context-aware, self-adaptive behavior of the applications, is modular, customizable and extensible so that it can by configured to better match the requirements and constraints of the individual applications.

1.3 Thesis statement

This thesis contends that traditional approaches for developing software applications are insufficient for meeting the software engineering challenges faced by the developers of context-aware,

self-adaptive systems targeting mobile and ubiquitous computing environments. It is suggested that the use of appropriate methodologies, facilitating software reuse in conjunction with middleware support, can render the development of context-aware applications significantly easier, faster and more cost-efficient.

The main contribution of this thesis is twofold: First, a development approach is proposed for designing and implementing context-aware applications as component frameworks. The constituent components realize the roles of context providers (referred to as context plug-ins) and context consumers (i.e., context-aware, self-adaptive applications), and can be reused across multiple platforms and/or shared by concurrently deployed applications. This methodology is further complemented by a *Model-Driven Development* (MDD) framework which extends reusability of context provider components to the finer data-structures and operators of which they are composed. Second, it proposes a modular middleware architecture which allows the developers to easily customize it and extend it, to better fit the requirements and constraints of the target applications and devices. Besides its wide applicability, this architecture features dynamic behavior which results in better resource utilization.

1.4 Approach

The research presented in this thesis has both theoretical and practical aspects. The former consists of a development methodology that enables the design of context-aware, self-adaptive applications using *Separation of Concerns* (SoC). This methodology is supported by a model for expressing contextual information and reasoning concepts. The methodology is also complemented by an MDD-based approach for developing reusable context plug-in components. The practical aspect consist of a pluggable and modular middleware architecture. This middleware

provides support for the deployment of context-aware applications—which are constructed using the proposed methodology—and facilitates context sensing, management and distribution.

1.4.1 Development methodology

The first contribution of this thesis is a methodology for developing context-aware applications, aiming for mobile and pervasive computing environments. The methodology adopts a component-oriented approach where the applications are split into *context providing* and *context consuming* components. In particular, specialized context plug-ins are used as replaceable and reusable components to *sense* or *reason on* (i.e., infer) context data. The methodology guides the developers throughout the initial requirements analysis, the development or reuse of existing plug-ins and, eventually, the deployment of the components required to realize the context-aware behavior of the application. The production of context-aware applications is supported by an elaborate *context model* which provides semantic consistency and enables advanced functionality such as inter-representation transformation of context data, and a *context query language* enabling conditional filtering of context information. Finally, the methodology is complemented by a model-driven development-based approach for the design and implementation of context plug-ins. In this approach, the plug-ins are first modeled in UML, and the model is then transformed to source code by a transformation tool.

1.4.2 Middleware architecture

The second contribution of this thesis concerns the design and implementation of a middleware architecture which is used to facilitate the deployment of context-aware applications. This middleware builds on top of the *Open Service Gateway initiative* (OSGi) framework, where the context providers (i.e., the context plug-ins) and the context consumers (e.g., the context-aware

applications) are defined as components (or *bundles* in OSGi terminology). In this architecture, both the context provider and the context consumer components register their context offers and needs with a central authority, the *context manager*. The latter manages the lifecycle of these components at run-time, while at the same time optimizing their resource consumption. Finally, the middleware features a modular architecture which allows quick realization of different variants of the middleware, depending on the requirements of the applications and the capabilities of the infrastructure.

1.5 Declaration and credits

The work presented in this thesis is, to the best of my belief, original and has been written by the author, except as acknowledged below. Also, I declare that the material presented in this thesis has not been previously submitted for a degree at this or any other university.

1.5.1 Additional publications relevant to the thesis

While a large part of this thesis consists of original work not yet published, a number of additional publications authored, or co-authored, by myself have contributed to the thesis in both direct and indirect ways.

The definitions for context, adaptation, variants and utility that appear in chapter 2, were originally presented in [80, 106] which were co-authored with Konstantinos Kakousis. The parts of the paper used in the thesis are the result of my personal work.

The related work presented in chapter 3 includes limited input from paper [27] which is a survey authored by Pyrros Bratskas and co-authored by myself and Konstantinos Kakousis.

The development methodology presented in chapter 4 is partially inspired by a previous work authored by myself [107]. Furthermore, the context model and the context query language, presented in the same chapter, are the results of joint work between the University of Kassel, Telecom Italia, Telefonica I+D and the University of Cyprus [114, 115]. This model builds on a more primitive one, developed earlier by the University of Cyprus in collaboration with partners from SINTEF, and evaluated by Integrasys [92, 93, 104, 108].

The pluggable and modular middleware architecture presented in chapter 5 was designed by myself. The resolution and activation mechanisms were originally presented in [110]. This paper was co-authored with partners from the University of Oslo and HP Italy. The University of Oslo contributed a method for integrating the proposed architecture with the MUSIC middleware and HP Italy helped in the experimental evaluation of the activation mechanism.

Furthermore, a number of papers authored or co-authored by myself discuss methods of context distribution but they only indirectly contribute to this thesis, by providing insight into the design of the context model and the middleware architecture. These publications [75, 76, 104, 108, 109] indirectly shaped the modeling support for context distribution presented in subsection 4.4.4, and the interface between the context distribution manager and the context middleware presented in section 5.4.

Finally, the entire chapter 6 is part of a joint work between the author and Roland Reichle and Michael Wagner from the University of Kassel. The authors share the idea for developing context plug-ins using a model-driven development approach, and they jointly developed the conceptual model. The author of this thesis contributed the implementation of the basic components and the case study evaluation. The partners from the University of Kassel contributed the UML profile and the transformation tools. At the time this thesis was submitted, the work in this chapter had not been published.

The papers I have authored or co-authored during the course of my doctoral dissertation were supervised by my research advisor, George A. Papadopoulos.

1.6 Structure of the thesis

The rest of this thesis is organized as follows.

Chapter 2 provides the foundations of this work. It defines the terms of context, adaptation, utility and self-adaptation. These definitions are then used to form a conceptual model for context-aware, self-adaptive applications.

Chapter 3 surveys the literature with respect to context awareness and self-adaptation, with emphasis in context sensing, modeling, management and development approaches. This chapter also surveys challenges and requirements identified in the literature, and examines specific solutions proposed by state-of-the-art approaches.

Building on the foundations of the previous chapters, a development methodology is presented in chapter 4. This methodology is based on a conceptual model which uses separation of concerns to simplify the application design and implementation. At the same time, this model also facilitates software reuse and together with a novel context model and a prototype context query language, they complement the development methodology.

Chapter 5 describes the middleware architecture developed to provide support to the development methodology. This middleware features a pluggable and modular architecture. The former allows it to automatically optimize the resource usage, as it is experimentally shown in the same chapter. The latter allows the developers to easily configure the middleware to the requirements of the specific deployment, thus better matching its capabilities.

While the development methodology separates context-aware applications to context providers and context consumers, the middleware architecture provides support for automatically handling

and optimizing their lifecycle. Chapter 6 presents a model-driven development approach which allows the development of the context providing components in a model-driven way, while also inheriting the benefits of model-driven development.

Both the development methodology and the pluggable architecture are evaluated in chapter 7. The evaluation was realized in three complementary phases. First, the detailed steps followed in the development of two case-study applications are presented, showcasing the advantages of the development methodology and the middleware architecture. Second, the results of a survey are presented, reflecting the feedback received from developers who used the proposed methodology and middleware. The latter include developers of context sensing and context reasoning plug-ins, developed as part of a research project, as well as students who used the methodology and the middleware as part an undergraduate course. Third, the requirements derived from the literature and listed in chapter 3 are revisited and the proposed solution is evaluated against them.

Finally, this thesis concludes with chapter 8. This chapter resumes the main contributions of this thesis to the state-of-the-art, and it lists a number of directions for future work.

Chapter 2

Foundations

This chapter aims to establish a common understanding of the basic concepts related to software development in general, and context-aware, self-adaptive applications in particular. In this respect, a few relevant definitions are first provided, followed by a conceptual model for context-aware, self-adaptive applications. These facilitate establishing a point of reference for understanding the problems investigated in this thesis. Furthermore, they facilitate understanding and evaluating the related work (see chapter 3), and also provide the foundation for the development methodology (see chapter 4) and the accompanying middleware architecture (see chapter 5).

The chapter starts with an introduction to the basic concepts discussed in this thesis, including *software engineering*, *development methodologies* and *software architecture*. Then, it presents the author's view on context-aware adaptation, from both a user and a system perspective. Mathematical-like definitions for context, adaptation, and autonomic behavior are then introduced, based on which it is shown how the middleware can be used as an enabling technology for context-aware adaptation. Finally, a high-level conceptual model for context-aware, self-adaptation enabling middleware is presented, serving as a benchmark for related approaches from

the literature. The chapter concludes with a discussion of how this model is used throughout the following chapters, as well as the limits of the research covered in this thesis.

2.1 Basic concepts

This thesis is concerned with the development of general context-aware, self-adaptive applications. Before these are discussed, some fundamental concepts used further on in this thesis are introduced. These include *software engineering* and *software architecture* in general, and *component orientation*, *model-driven development* and *middleware* in particular.

2.1.1 Software engineering

Software engineering is defined as "*the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches*" [17]. The term has originally appeared in the *1968 NATO Software Engineering Conference* and was meant to provoke thought regarding the *software crisis* that the software community was facing at the time [96].

Software engineering is important because the production of software is expensive. Arguably, the software production cost increases as a function of its complexity. In this regard, and similar to other engineering principles, software engineering aims to provide a disciplined approach for producing software efficiently and with minimal cost. Arguably, highly complex software systems would not even be possible without fundamental software engineering principles and concepts, like *modularization* [102] and separation of concerns [62, 83]. These approaches employ the fundamental principle of solving a problem by breaking it down to smaller and simpler sub-problems.

2.1.1.1 Component orientation

Component orientation is a software development approach where the targeted software system is decomposed into individual entities, each realizing a specific concern. These entities are represented as *components*. Widely cited, Szyperski's definition specifies software component as: "[...] *a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition*" [131].

Arguably, component-orientation provides important benefits. Beyond separation of concerns (see subsection 2.4.3), it also promotes software *reusability*. This is facilitated because, as indicated in the above definition, components provide contractually specified interfaces thus enabling independent development (i.e., individual development teams can deal with the creation of various components, both providers and consumers of the commonly agreed interface).

2.1.1.2 Model-driven development

Model-driven development (MDD) is the process of developing software systems via an iterative process, where the models abstracting some particular domain concepts are first defined (rather than the computation or algorithmic concepts) before they are transformed to source code using some appropriate transformation scripts (see section 6.1). Specially designed tools are used to enable the modeling of the targeted system architecture and its transformation into code (i.e., computer-based tools that are intended to assist the software life cycle processes [17]).

In [127], Schmidt argues that model-driven engineering (a synonym of MDD) provides the means to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively. The author of this thesis shares this opinion and

contends that the use of MDD technology can provide significant benefits in the process of constructing domain-specific functionality, such as the context-aware behavior of mobile applications.

2.1.2 Development methodologies

In order to enable cost-efficient production of applications, software developers resort to structured and well-defined processes, also referred to as *development methodologies*. In this way, rather than following an impulsive and often undisciplined approach, software developers proceed according to a planned set of steps allowing them to control and mitigate development complexity.

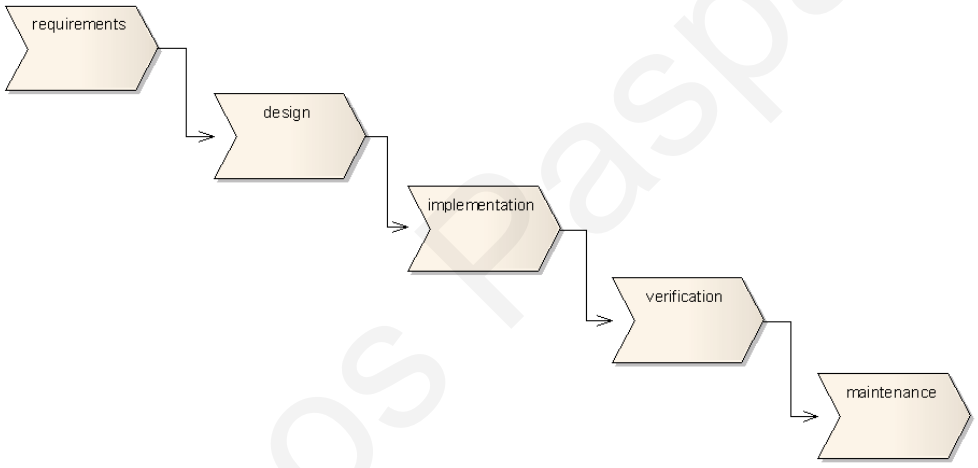

Figure 1: A waterfall software development model

It is worth noting that the development methodologies proposed in the literature, and applied in practice, share a common characteristic: they all break the development task down to a number of subtasks realizing the software's life-cycle. These subtasks include *conceptualization*, *requirements analysis*, *design*, *development*, *testing*, *deployment*, *maintenance*, etc.

Individual development methodologies propose different ordering of the subtasks and, also, elaborate flows for the sequence of their constituent steps. A classic development methodology is the *waterfall model*, illustrated in figure 1.

In this model the developers follow a predefined set of steps which enables them to better plan and analyze the implementation of their applications. The waterfall model is naturally considered a rather inflexible approach, as it requires for each step to be thoroughly completed before proceeding to the next one. For instance, the developers first provide the requirements specifications, which is then followed by the system design, which then is followed by the software implementation, etc.

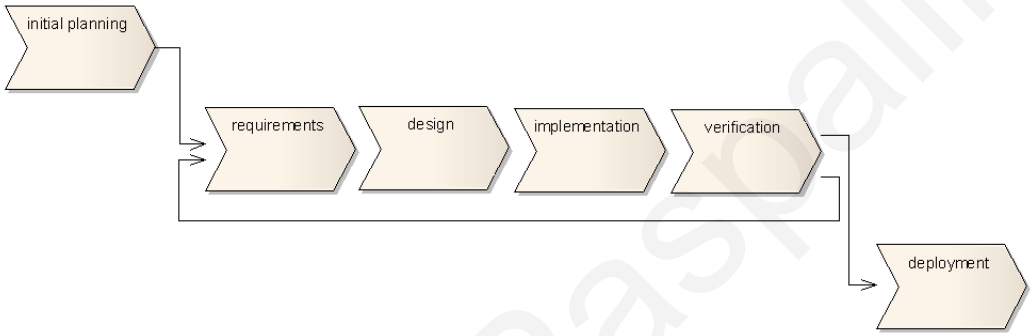

Figure 2: An iterative software development model

Alternative methodologies follow different approaches, and some include loops in their control flow. For instance, iterative and incremental development models specify cyclic development processes where after an initial planning phase, the development proceeds with a number of cycles, iteratively repeating the corresponding tasks. In each of these cycles, the system is further planned, implemented, tested and evaluated before it is, eventually, deployed. Figure 2 illustrates an example of the iterative development model. Many more models are also widely used by enterprises, such as the V-model methodology, which is considered an extension of the waterfall model.

2.1.3 Software architecture

Stafford and Wolf define the *software architecture* of a system as "*the arrangement of its components into one or more structures defined by the functional role played by each component and the interaction relationships exhibited by the components*" [65].

Similar to buildings architectures, which are best envisioned in terms of a number of complementary views or models, so too are software architectures. Most notably, *structural* views help the developers to document and communicate the basic components of a system, along with their interactions. Furthermore, *behavioral* views help the developers visualize and analyze how components interact to accomplish their preassigned responsibilities.

2.1.3.1 Middleware

Many modern software systems define middleware-based architectures. Formally, middleware is defined as "*reusable software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware*" [123]. Traditionally, the role of middleware was to hide resource distribution and heterogeneity from the business logic of applications. However, middleware technology has proved to be quite versatile, thus extending its role to a more general domain: "*bridging the gap between application programs and the lower-level hardware and software infrastructure to coordinate how parts of applications are connected and how they interoperate*" [60].

Schmidt decomposes middleware into four distinct layers according to the functionality they are designed to offer: *host-infrastructure middleware*, *distribution middleware*, *common middleware services* and *domain-specific middleware services* [126]. A common, specialized domain of middleware-based systems includes mobile computing, as discussed in a survey by Mascolo et al

[88]. This thesis focuses on domain-specific middleware architectures which facilitate the development of context-aware, self-adaptive applications. These architectures are discussed in section 2.5, and a specific middleware architecture implementation is presented in chapter 5.

2.2 Context-aware adaptation

A fundamental property of context-aware systems is their ability to autonomously—and often proactively—adapt their behavior by utilizing their knowledge on context. Naturally, their behavior is adapted to serve a predefined goal. This thesis is primarily concerned with mobile and embedded devices, deployed in pervasive computing environments, where the predefined goal is to *optimize the end-user experience*. In other words, the context is sensed and the adaptations are decided with the purpose of optimizing the *utility* as that is perceived by the end-user in the mobile or pervasive computing environment.

2.2.1 The user perspective

It is argued that users experience the utility of a service, as it is offered by a context-aware application, by interacting with the corresponding service provider. For instance, consider an on-site technician who uses her *Personal Digital Assistant* (PDA) to access information on her dynamically updated schedule while visiting client sites. In this case the provided service utilizes a distributed system where the technician's PDA acts as the service client and a centralized server acts as the service provider. It is argued that in addition to the functional aspects of this particular service, the technician also experiences its extra-functional behavior. For instance, the user perceives the data richness of the service (e.g., whether high-quality or low-quality images are attached to her assignments), as well as the network latency and bandwidth (e.g., how long it took for the PDA to get synchronized).

Regardless of whether the effectiveness of a service is measured by the existence of some perceived results, or the lack of such as per the ubiquitous computing paradigm, a service can be analyzed in two parts. The first part refers to the *functional behavior* of the service, or simply delivering what the service was originally designed for. In the previous example, the functional behavior refers to the ability of the technician to dynamically access and update her schedule. As it has been already argued though, real world applications are also characterized by what is perceived by users as *extra-functional* behavior. This kind of behavior, especially in the context of mobile and pervasive computing environments, is generally unpredictable because it is affected by numerous exogenous factors such as the occupation and position of the user, the network availability, the light conditions, etc. For example, the quality of the attached images and the network latency/bandwidth are examples of factors affecting how the user perceives the application's extra-functional behavior. Arguably, even if the user does not consciously think of these extra-functional properties, they nevertheless affect their overall experience and, consequently, their satisfaction with the service.

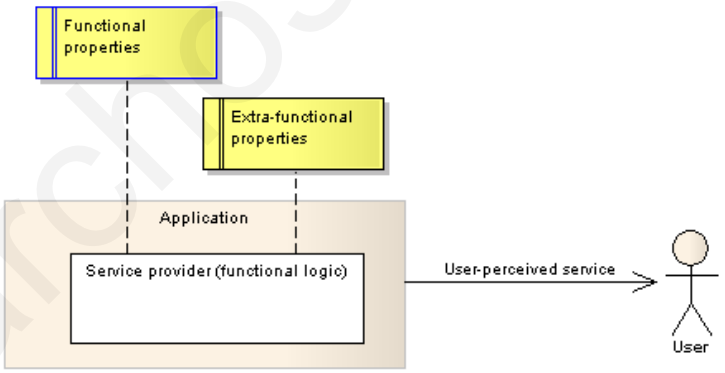

Figure 3: User perspective: the combined result of the application logic and its extra-functional behavior

As it is illustrated in figure 3, the user perceives the service as the combined result of both the application logic (or business logic) and that of its extra-functional behavior [107]. This thesis proposes a software development methodology which enables the separation of the two concerns:

developing the application logic and defining its context-aware, self-adaptive behavior. With this approach, it is argued (and shown in chapter 7) that the developers are enabled to concentrate on the functional and extra-functional requirements of their project independently, rather than mixing the two concerns in the same phase, which results in improved productivity. The business logic of an application is designed and implemented as per the component-orientation paradigm [65, 131] where individual functional aspects of the application are realized as components. Following that, the context-aware and self-adaptive behavior are treated as additional aspects of the application, which are independently designed and realized.

2.2.2 The system perspective

In addition to the user perspective, context-aware, self-adaptive systems can also be viewed from a system perspective, as illustrated in figure 4. In this view, the device hosting the context-aware, self-adaptation logic interacts with the execution environment by *sensing* its properties and reacting accordingly in order to *shape* it.

Figure 4: System perspective: sensing and shaping the environment

From this perspective, the device hosting the context-aware, self-adaptation logic is viewed as one entity and the execution environment as another. The execution environment includes anything that can affect the interaction of the user with the intended application (i.e., all the properties of

the environment in which the interaction takes place, including the user and the device themselves as per Dey's definition of context [44, 45]).

Assuming that a system consists of these two entities, then its behavior is summarized by two basic concepts: *sensing* and *shaping* the environment. In the sensing direction (upper arrow), the autonomous entity collects as much information as possible about the execution environment. This is used in the shaping direction (lower arrow), where the environment is changed by applying decisions that were taken by the autonomous context-aware, adaptation logic.

2.3 Definitions

Based on the user and system perspectives, this section introduces formalized definitions for the basic concepts used in this thesis: *context*, *adaptation* and *autonomic behavior*. Whenever possible, mathematical-like definitions are provided, as it was originally proposed in [106], to enable a more accurate model of the context-aware, autonomic behavior.

2.3.1 Context

As it was already mentioned in section 1.1.2, this thesis adopts Dey's definition of context [44, 45]: "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves*".

In practice, context can be divided into several orthogonal types, or dimensions. Some of these are infinite (e.g., time) while some others are bounded (e.g., the *user's gender* can either be "male" or "female"). In this perspective, the context might be modeled as a multidimensional space in which each modeled context type defines a dimension (in the case of types with finite

value-domains, each value is assigned a range in the dimension). Similar approaches exist in the literature, such as the one presented in [100].

Assuming that each context type can be abstracted by a real number (i.e., $\mathbb{R}$), then a context space of d types can be abstracted as a d-dimensional space $\mathbb{D}$. Then, at any time t, the context can be represented by a point c_t, which defines a value for each of the d dimensions (i.e., the c_t is defined as $(c_t^1, c_t^2, ..., c_t^d)$, where c_t^i indicates the context value at dimension i for the time instance t). These points (i.e., c_t) are referred to as *context instances*. The context space and the context instances are more formally defined as follows:

Context space
context space $\equiv \mathbb{R}^d$.

Context instance
$c_t \equiv (c_t^1, c_t^2, \ldots, c_t^d) \in \mathbb{R}^d$, where $c_t^i \in \mathbb{R}\ \forall i$ in $[1..d]$.

In these definitions, the *context space* (or simply context) is defined as a d-dimensional geometric space, and a *context instance* is a point (or vector) in the space. Although a geometric analogy is used to characterize the context, no assumptions are made concerning the relation between points, especially their geometric distance. Since discrete states of context can be mapped to (arbitrary) real values, no relation can be guaranteed for neighboring points.

2.3.2 Adaptation

Adaptation is a process by which software changes its behavior in order to better match the changing environment and the user state. It is suggested that pervasive computing and autonomic computing are the two main forces driving the proliferation of software adaptation technology [90]. In the context of mobile and ubiquitous computing environments, software adaptation is required to overcome the variability which is inherent in such environments. In this respect, systems are designed with adaptive properties so that a system can be configured in different modes (i.e.,

combinations of component compositions and parameter settings), each of which is designed to maximize the utility for at least some points in the context space (see subsection 2.3.1).

Software engineering, defines two basic approaches for software adaptation: *parameter-based adaptation* and *compositional adaptation* [90, 91]. The former refers to changes to a parameter (like the polling interval of a context sensor) and the latter refers to more intrusive changes to the system's architecture, possibly adding new—or, perhaps, updated or corrected—behavior (like a new codec realizing faster encoding). In this thesis it is assumed that the developed applications are subject to both parameter-based and compositional adaptation. In the first case, adaptation is achieved at the level of individual components, which export appropriate APIs for controlling one or more input parameters. In compositional adaptations, the architecture of the application is defined so that specific *roles* in the application can be fulfilled by any of a set of alternative component realizations [50].

While the variation points in the architecture of an application are defined at design-time, the actual alternative realizations are computed at run-time. These alternatives are called *variants* (a more formal definition of variants is provided in the following paragraphs). A fundamental property of the alternative variants is that they maintain the *functional properties* of the role they realize, while varying its *extra-functional* properties. In this case, the purpose of the context-aware, self-adaptive system can be seen as the selection of a variant appropriate extra-functional properties aiming to optimize the perceived utility [107].

For instance, assume that a particular application can be composed of a finite set of N alternative variants as follows: $\{variant_1, variant_2, \ldots, variant_N\}$. In practice, the set of variants can be infinite such as, for instance, when the value domain in a parameter-based adaptation is unbound. For example, consider a component which can be adapted dynamically by setting an internal parameter to any value within the range [0, ∞] (i.e., "any positive integer"). In practice, in

order to simplify the analysis of such cases it is assumed that the adaptation domain is transformed to a finite set of configurations by quantizing their value range (i.e., by mapping ranges of the infinite domain to a finite set of value ranges). For instance, continuing with the previous example, the "any positive integer" parameter can be quantized to just three values as in {"0", "greater than zero and less than 10", "10 or more"}. Naturally, the exact selection of ranges is preferably chosen in such a way so that the quantization does not exclude *important* variants. In this case, important means variants which are significantly more suitable than others in certain context conditions. This way, it can be guaranteed that the number of application variants is always finite.

In some cases, these variants are defined a priori by software developers. However, in order to allow for maximum flexibility and also to meet the requirements of highly dynamic environments, the variants are often required to be formed dynamically at run-time. For instance, in component-based systems the variants can be formed by examining the provided and required services of each component (i.e., interfaces) [36]. Given these dependencies, it is possible to realize mechanisms that form variants as compositions of components where their dependencies are all resolved. Naturally, the exact set of available variants fluctuates according to the availability of components and services and also according to the context. In this respect, the definition of an application variant is summarized as follows:

> **Variant**
> A variant is any parameter-based or compositional-based configuration of the application, maintaining its original functional properties.

As it was already argued, a set of all possible variants for an application can be assumed to be always finite. Such a set of variants can thus be denoted by a set as follows:

> **Variants**
> $variants \equiv \{variant_1, variant_2, \ldots, variant_N\}$.

This thesis is primarily concerned with component-based applications, and thus it assumes that the system comprises either a single application or a set of applications. However, the utility

of each individual application is considered separately, partly based on the user preferences for each one of them, as it is discussed in the following section.

2.3.3 Self-adaptive, context-aware behavior

Having defined *context* and *adaptation*, the next important concept is that of *self-adaptive, context-aware behavior*. This concept refers to the ability of a software system to sense the environment and *autonomously react* to that stimuli in order to shape it. This sort of behavior is similar to the *Sense-Plan-Act* (SPA) architecture used in robotics [28], and is also extensively studied in the field of *Artificial Intelligence* (AI) [135]. While the state of the art includes alternative approaches for enabling autonomic adaptation decisions, in this thesis a *utility function*-based approach is used. This decision is justified by the fact that utility functions fit well with variant-based adaptation approaches discussed in the previous subsection and also because they accommodate changes to variants availability at run-time (an important feature in highly dynamic environments).

For example, consider a user in a mobile or pervasive computing environment. Such environments are generally designed to offer services to the users and they involve both direct and indirect user interaction. In both cases it is assumed that the user experiences the service and has a personalized opinion about its utility (i.e., different users perceive the utility of the same service differently). In this discussion, the *utility* refers to a quality metric, broader than *Quality of Service* (QoS), which aims to capture the general user satisfaction with the functioning of a system in a given context. For instance, if a user prefers a system configuration over another one, then it is assumed that the former has a higher utility.

While utility is highly conceptual, it can be partly quantified using mathematical notions. Here, *utility functions* are referred to as mathematical artifacts which map combinations of context states and variants to scalar values, typically in the range $[0, 1]$ where 0 indicates minimum (worst)

utility and 1 indicates maximum (best) utility (i.e., quite similar to the notion used in micro-economics where utilities represent user happiness [101]). The choice of the $[0, 1]$ bounds provides the convenience of allowing the multiplication of different utilities without exceeding the original bounds. The basic purpose of a utility function is to provide a formal, mathematical method for computing the utility of a service, as it is perceived by the end-user. In this respect, utility functions are defined as follows:

> **Perceived utility**
> Perceived utility ($U_{perceived}$) is a function that for any context point c_t, it maps any two variants $variant_x$ and $variant_y$ to scalar values (e.g., in the range of $[0, 1]$) so that $f(c_t, variant_x) > f(c_t, variant_y)$ if and only if the user prefers $variant_x$ to $variant_y$ from her or his point of perception (noted as $variant_x \succeq variant_y$).

Given this definition, the problem of decision in self-adaptive context-aware systems becomes the formation of a computed utility function ($U_{computed}$) which can approximate the perceived utility.

Evidently, the main feature of a utility function f is that it enables the computation of a utility value for each possible variant V_i, thus enabling total ordering of variants for a given context c_t. This is illustrated in the following:

> **Utility-based total ordering of variants**
> $Total\ ordering(f, c_t, variants) \equiv (variant_{i_1} \succeq variant_{i_2} \succeq \ldots \succeq variant_{i_N})$
> where the utilities of the variants for the given utility function f and context c_t satisfy the inequality: $f(c_t, variant_{i_1}) \geq f(c_t, variant_{i_2}) \geq \ldots \geq f(c_t, variant_{i_N})$

In this definition, the $\succeq$ operator is used to denote (relatively) better suitability of a variant to a given context. As far as adaptation is concerned, an autonomous context-aware adaptive system should aim at always selecting the most suitable variant as context changes.

2.3.4 Context-awareness versus self-adaptiveness

At this point, it should be disambiguated how context-aware applications are viewed, in relation to self-adaptive ones. From the perspective of this thesis, context-aware are the applications

that use contextual information at run-time. Some applications, however, use context information simply to complement their *functional* behavior. For example, a mapping application running on a mobile device often uses location information—for instance, offered by an embedded *Global Positioning Sensor* (GPS)—in order to automatically center the map view at the current location when launched. These applications are viewed as *purely context-aware*.

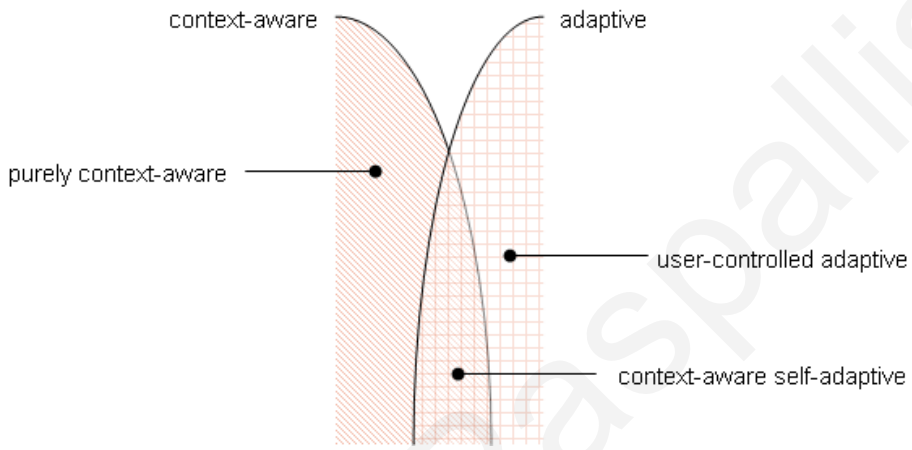

Figure 5: Classification of context-aware applications

On the other hand, some context-aware applications use context information primarily to adapt their *extra-functional* behavior. For example, an application offering two modalities—one with *low* and one with *high* power consumption profiles—can be adapted accordingly based on the remaining battery level. The main difference, compared to context-aware applications, is that the change in the application affects mainly the extra-functional behavior. For instance, the low-power modality might offer less frequently updated data which could be preferable in situations where battery endurance is more important. These applications are viewed as *context-adaptive*.

Regarding *adaptive* applications, there are mainly two classifications: those which are self-adaptive and those which are explicitly adapted by an external entity such as a user. This thesis is concerned with *self-adaptive* applications only. In this type of applications, the self-adaptation is usually controlled by a feedback loop which takes as input some relevant stimuli. Because

a quite general definition of context is adopted, all *self-adaptive* applications can be viewed as *context-adaptive*, meaning that the stimuli can always be considered as a subset of context.

This classification is illustrated in figure 5. The super-set of context-aware applications consists of those applications which are purely context-aware and those which are context-adaptive. Naturally, some applications can partly act as purely context-aware (i.e., using context information in their functional-logic) and partly as *context-adaptive* (i.e., also using context information in their extra-functional behavior). This thesis aims at facilitating the development of both kinds of context-aware applications.

The question of categorizing context-awareness has been a long running topic, as it is evident in the literature. A detailed discussion has first appeared in Dey's thesis [44], where it is noted that definitions of context-aware computing fall into two categories: *using* context and *adapting* to context. The classification presented in this subsection and, illustrated in figure 5, is consistent with both this categorization, as well as with Dey's own definition of context-awareness: "*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task*".

Further relevant definitions have also been proposed. For instance, Noble defined *application-aware adaptation* as a collaborative model where both the system (i.e., the operating system) and the application share the responsibility of adaptation. The former manages the resources—as it is in the best position to determine resource availability—while the latter specifies the adaptation policy—as it is the only entity that can properly decide how to adapt to a given situation [98].

2.4 Middleware as an enabling technology

This section argues that middleware-based approaches add significant value to the process of developing context-aware, self-adaptive applications. A similar statement was presented in [91],

where it was also argued that the confluence of *component-based design*, *computational reflection* and *separation of concerns* aids the realization of self-configuring systems. Here this argument is extended, and it is shown how custom-tailored middleware (see subsection 2.1.3.1) aid the development of context-aware, self-adaptive applications.

2.4.1 Component-based design

Component-orientation evolves object-orientation by enabling transparent software reuse. One of the main characteristics of software components is that they provide well-defined interface specifications which act as *contracts* when different components of a system are individually designed and implemented by individual teams. Furthermore, as components are often defined as black boxes (i.e., their internal realization details are hidden), such contracts enable their inclusion in applications simply by examining their external API (i.e., required and offered interfaces).

The literature includes a large number of composition mechanisms, both static and dynamic. An extensive taxonomy of compositional adaptation was presented by McKinley et al [91]. This thesis is concerned with dynamic adaptation which is decided and executed at run-time. This is necessary because of the dynamism which characterizes mobile and pervasive computing environments. This dynamism results in a continuous stream of context changes, which in turn require real-time evaluation and response by adaptive systems aiming to continuously optimize their operation.

2.4.2 Computational reflection

Computational reflection refers to the ability of a program to reason about and possibly alter its own behavior [87]. Reflection enables both *introspection* and *intercession*. Introspection refers to the ability of a program to reason on its internal behavior and intercession refers to its ability to

alter it. Reflection is further classified into *structural* and *behavioral*, where the former addresses issues related to class hierarchy and object interconnections (e.g., querying the available methods of an object) and the latter focuses on computational semantics of the application (e.g., selecting and loading a specific network protocol) [91].

In many modern programming languages, such as in Java [22], reflection is a design feature of the language. This kind of *systematic*, as opposed to *ad hoc*, introspection and intercession are often enabled with the use of a *Meta-Object Protocol* (MOP) [82]. In that case, an interface is used to provide well-defined methods for both structural and behavioral reflection.

2.4.3 Separation of concerns

The concept of *separation of concerns* is generally defined as the process of breaking the development of a software system into multiple features, with as little overlap as possible. The term was used as early as in 1974 by Dijkstra [48] and continued becoming more popular as it is evident by the inclusion of the term in Chris Reade's book 15 years later [113]. This thesis uses separation of concerns as a fundamental method for designing and implementing context-aware, self-adaptive applications [107]. In this regard the context-aware and the self-adaptive behavior of the application are both considered as individual concerns, cross-cutting with its business logic.

As it was illustrated in figure 3, the user perceives the combined effect of both the functional and extra-functional aspects of the application. For instance, referring back to the technician's example (see subsection 2.2.1), the user experiences the utility of the *schedule service* as the aggregate of the functional features of the application (e.g., being dynamically notified of new assignments) and its extra-functional properties (e.g., the time it takes to synchronize new assignments). Extending this example to the development methodology, it is argued that it is possible to

separate the development of such an application into two concerns: First, developing the business logic of the application and, second, realizing its context-aware, self-adaptive behavior.

2.5 Domain-specific middleware for context-aware, self-adaptive applications

Given the definitions of section 2.3 and the guidelines of section 2.4, a middleware-based, conceptual model for context-aware self-adaptive applications is defined. This model provides the foundation for the development methodology presented in chapter 4.

As it has been already argued, context-aware adaptive applications are assumed to be designed and developed as compositions of components. It is further assumed that the availability of components at run-time is dynamic, either because some components are updated, new ones are added or existing ones are removed. Naturally, the dynamic availability of components affects the availability of application variants as well. For instance, removing a component M_j renders all variants depending on it as obsolete.

As it was already discussed in section 2.4, one of the possible roles for middleware-based solutions is to serve *reusable, domain-specific functionality*. Here, such middleware is implicitly defined as a custom-tailored design which enables context-aware, self-adaptive applications, by defining the following set of four requirements.

2.5.1 Domain-specific middleware requirements

- *Component support*: The middleware should provide support for installing, uninstalling, and updating components, as well as for managing their lifecycle. In order to support run-time adaptation, the component framework should also support dynamic reconfiguration.

- *Application composition*: The middleware should provide support for dynamically forming applications as component compositions. The exact methods for composition, and thus the set of possible variants, depends on the underlying *component support* middleware.

- *Context management*: One of the domain-specific functionalities desired in this middleware is *context management*. This functionality facilitates coordinated collection, storage and processing of context information, which arguably results in better resource utilization.

- *Adaptation reasoning*: Ultimately, in order to enable autonomous self-adaptation of applications based on contextual information, the middleware must be capable of reasoning based on the context state, and adapt the deployed applications accordingly in order to optimize the end-user utility.

2.5.2 Conceptual model for context-aware, self-adaptation enabling middleware

Based on these requirements, a conceptual model is defined for a middleware supporting the deployment of context-aware, self-adaptive applications. This model breaks this domain-specific middleware into four layers, in line with the detected requirements, as illustrated in figure 6.

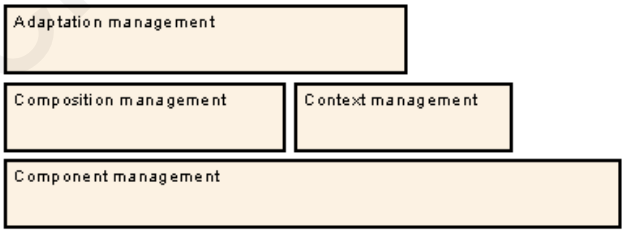

Figure 6: A conceptual model for domain-specific middleware enabling context-aware, self-adaptive applications

First, the *component management* layer is responsible for managing the individual components of the deployed applications. As the middleware aims for *dynamic* adaptation, this layer is

responsible for enabling the installation, uninstallation, activation, deactivation, binding and unbinding of components, all at run-time. In this layer, the components are viewed as individual entities, thus hiding any details related to their *role* in specific applications. Furthermore, it is assumed that the deployed components conform to a predefined standard, exporting an API that allows their control by the middleware using the *Inversion-of-Control* (IoC) paradigm [99].

Second, the *composition management* layer is responsible for forming the possible compositions of each application. In practice, this means that given the set of available components (in coordination with the underlying, component management layer), this layer is responsible for computing the set of possible application compositions (i.e., form the set of $Variants$, as defined in section 2.3.2). It should be noted that when multiple applications are deployed concurrently, then the formed compositions might define *shared* components when suitable. While most component-based applications are primarily designed so their component architecture is adapted at compile or load-time, this thesis is primarily concerned with enabling dynamic composition at run-time. This enables the development of applications which are capable of adjusting their behavior (both in terms of functionality and/or quality) by dynamically installing and activating new components.

Next, the *context management* layer is responsible for centrally managing the collection, storage, processing and filtering of context data. In other words, this layer is responsible for making the context information available to the deployed applications and the other middleware layers (notably, the adaptation management layer). Such a centralized approach provides significant advantages. Most notably, individual applications requiring the same context information do not need to replicate the same code individually, but rather they can reuse the same functionality as it is provided by a centralized authority.

Finally, the *adaptation management* layer is responsible for reasoning based on context information and for selecting the most appropriate composition for each of the deployed applications. This thesis is concerned with user-centric adaptation, and thus this layer aims at optimizing the end-user experience by selecting those variants that maximize the user-perceived utility (i.e., select $variant_j$ so that $variant_j \succeq variant_i \; \forall \; i, i \neq j$). The selected composition is eventually applied with the help of the composition and component layers.

It should be noted that the higher middleware layers utilize services from layers below them, while at the same time enabling direct interfacing of applications with all layers. This is necessary, as applications are component-based, and thus require the component-layer to control their lifecycle. Also, the composition layer needs to have direct access to the applications too, so that it can collect information concerning the possible bindings between components in order to form the set of possible variants. Finally, the context management layer is also made directly accessible to applications, so that they can explicitly inquire context information which is related to their functional logic (e.g., access location information in order to center a map).

Arguably, this conceptual model can form the basis for the development of context-aware, self-adaptive applications. This conceptual model builds on *component-based design*, *computational reflection*, and *separation of concerns* as important building blocks of the middleware.

Component-based design is a fundamental assumption for this middleware, as component-orientation greatly facilitates compositional adaptation. Second, computational reflection, in terms of dynamically inferring the ways in which the components can be composed to form the applications, is also required for the formation of the application variants. Finally, with respect to separation of concerns, it is argued that this conceptual model enables separation of the context-awareness and self-adaptation concerns from that of the application business logic.

2.6 Discussion

This chapter has introduced the basic concepts that this thesis is dealing with. Starting from a high-level point of view, it discussed software engineering, development methodologies, and software architectures. Next it provided the author's view on context-aware adaptation from both a user and a system perspective. This was followed by formal, mathematically-backed definitions of context, adaptation and utility. Based on these definitions, a middleware architecture blueprint was proposed. Finally, this chapter presented a conceptual model which defines a domain-specific middleware architecture, enabling context-aware adaptive applications while also enabling separation of concerns for the *component and composition management*, *context sensing* and *adaptation reasoning*. This conceptual architecture allows the deployment of context-aware, self-adaptive applications that require limited (or none at all) explicit interaction with the middleware. Rather the middleware automatically and autonomously controls the lifecycle and context-aware, self-adaptive behavior of the application using the *Inversion-of-Control* paradigm.

In this chapter, it was assumed that all context-aware, adaptive systems specify alternative run-time variants (either explicitly or implicitly). However, in many of the context-aware frameworks and development approaches described in the literature (see chapter 3), this assumption does not immediately hold. For instance, many of them use either the *triggering* or the *branching* programming model [66, 68]. Nevertheless, it is argued that even in such alternative approaches, run-time variants can be implicitly defined. For instance, in the triggering approach, certain context events are used to fire the corresponding actions which, as a result, either print results in the display or adapt the application's logic (e.g., by selecting to switch a wireless network adapter on or off), thus implicitly defining a variation point. In the case of the branching programming model, the

correspondence to variation points is even more explicit. Certain flows of application logic are selected and followed as a result of certain context values, thus explicitly defining variation points. By definition, the set of all variation points, along with their value domain, specify the run-time variants of the application.

While this chapter has introduced a conceptual model covering both aspects of context-awareness and self-adaptation, the main results of this thesis are limited to context management only. This refers to models, methods and tools which are responsible for collecting, storing, processing, and distributing context information with the purpose of making it readily available for use by context-aware applications. As it was discussed in subsection 2.3.4, by adopting a rich definition for context, the class of self-adaptive systems can be considered as a subset of context-aware systems. Based on this assumption, the approach discussed in this thesis proposes a methodology and a middleware architecture for developing general context-aware, self-adaptive applications but it does not detail methods for explicitly defining variation points or composition plans (these concepts are extensively described in [50, 54, 116]). Instead, the models, methods and tools described in the rest of this thesis focus on the context management problem, and in particular on ways to ease the development of general context-aware systems. For these, both stand-alone context-aware applications, as well as more complex middleware systems, functionally dependent on context information, are considered.

Chapter 3

Related work

Context-aware computing has been actively studied since at least mid-nineties [125], shortly after Weiser first introduced the concept of Ubiquitous Computing [139, 140]. Initially, the research was concentrated on stand-alone context-aware applications, especially location-aware ones such as tour guides and active badges [86, 103, 137]. However, as the interest for context-awareness was extended to cover a much wider domain of context types and uses, the research was naturally shifted towards building *frameworks* providing support for general context-awareness. Perhaps the most popular of those is the Context Toolkit [46, 44] while, although more recent, approaches like CML [68, 70] and COSMOS [41, 117] are also widely referenced.

This chapter discusses the current state-of-the-art with regard to research on context-awareness. First, the challenges related to developing context-aware systems are studied, followed by an extensive list of requirements as they are documented in the literature. These requirements are further discussed and used in the rest of this thesis as they provide a well-formed benchmark for the design of the development methodology and the middleware architecture presented in this thesis. Finally, a number of representative context-aware frameworks are presented, demonstrating the different

approaches followed in the literature. Some of these approaches are revisited in chapter 7 where they are compared to the approach described in this thesis.

3.1 Challenges

Some definitions of context were already presented in subsection 1.1.2. This section provides a view of the challenges related to the development of context-aware systems, as they were identified in part of the literature.

Despite the fact that researchers have attempted to meet the challenges of engineering context-aware systems since the early nineties, still not many commercial applications and products are widely available. Many researchers argue that this is mainly due to the significant complexity that is inherent in the development of such context-aware systems [66] while others argue that missing common standards and social barriers are equally important [142]. The following list includes challenges that have been published in the literature, reflecting how researchers view the problem of enabling context-aware applications. While extensive, the following list is not exhaustive.

- In 1997, Pascoe argued that "*... the primary reason* [for the slow adoption of context-aware applications] *is the difficulty and lack of any supporting infrastructure in capturing and processing context data, so hampering the development of context-aware systems*" [103].

- A few years later, Dey, Abowd and Salber contended that "*... one of the main reasons why context is not used more often in applications is that there is no common way to acquire and handle context. In general, context is handled in an improvised manner*" [46].

- As it was discussed in subsection 1.1.1, a set of four challenges were identified by Satyanarayanan as *proposed research thrusts* [122]: effective use of smart-spaces, invisibility, localized scalability and masking uneven conditioning. While these challenges were not

explicitly stated for context-aware computing, nevertheless two of them are of particular relevance to its success.

First, *localized scalability* refers to the need for controlled interactions between entities in a smart-space environment. With regard to context distribution, for example, it is required that appropriate methods are used to ensure context sharing is limited to a subset of interested parties only [108]. Just like the inverse square law of nature, good system design should aim to achieve scalability by severely reducing interactions among distant entities.

Second, *masking uneven conditioning* refers to the challenge of dealing with heterogeneous environments, where the penetration of pervasive computing (with support for context-awareness) varies considerably, depending on both technical and non-technical factors (such as organizational structure, economic and business models, etc). In the interim of globally even conditioning (if it is ever achieved), the smartness of different environments will greatly vary. Successful systems should be able to automatically compensate for that variability (e.g., by providing off-line access in cases where there is no network coverage). From a context-aware perspective, the underlying frameworks should be able to compensate for varying conditions affecting the collection, processing and dissemination of context information (e.g., use an alternative location sensor, thus bypassing GPS while indoors).

- In [40], Cheverest et al argue that some potential pitfalls might arise from adapting to context-aware applications: First, it should be ensured that the system is not "*failing to reach a stable state*". That is, in case both the user and the system attempt to adapt to the current context, then it might be impossible for the system to reach a stable state. Second, the developers should aim at finding the right balance in "*the tradeoff between autonomy and user control*". Also, the user should be able to "*trust the agent performing the adaptation*".

- In her PhD thesis [66], Henricksen argues that the reason there are no commercially successful examples of context-aware applications is associated to [the difficulty imposed by] three challenges: "*the analysis and specification of application requirements for context information, the acquisition of the required information from suitable sources—such as sensors—and subsequent management and dissemination of relevant information to applications, and the design and implementation of suitable context-aware behaviors that meet the unique usability requirements of pervasive computing environments*".

  Also, in [68], Henricksen and Indulska state that despite the flurry of interest in context-awareness, such applications have still not made it out of the laboratory and into our everyday lives. They argue that this is largely due to the "*high application development overheads, social barriers associated with privacy and usability, and an imperfect understanding of the truly compelling uses of context awareness"*. Similar concerns regarding the hurdles before widespread adoption of context are expressed again in [70].

- In a more recent article in the *Communications of ACM* [142], information architect Alex Wright predicts that "*the next generation of smart-phones will take advantage of the growing availability of built-in physical sensors and better data-exchange capabilities to support new applications that not only keep track of your personal data, but also track your behavior and anticipate your intentions*".

  However, before this vision is realized, Wright identifies two important challenges. First, "*the lack of open standards for exchanging context data between applications*", which results to numerous competing operating systems (OS) and standards (e.g., Bluetooth, UPnP,

ZigBee, etc), must be overcome. Second, Wright states that "*beyond the technical and business hurdles, perhaps the greatest challenge will come from the most unpredictable variables of all: users*", referring to the challenge of protecting user privacy and thus gaining their trust.

While this is not an exhaustive list of the challenges in enabling context-awareness, it is nevertheless indicative of the research work performed so far. Even challenges identified in the earliest research works are still valid—and probably the most notable ones—preventing widespread adoption of context-aware applications. These are related to development complexity and are complemented by the increasing concerns regarding user privacy, especially when sensitive information like *user location* is made available to applications and services.

While these challenges are all valid and representative of the reality faced by the developers of context-aware applications, they are rather high-level, reflecting only a bird's-eye view of the complete set of the associated difficulties. The next subsection goes one step further, by listing a comprehensive list of technical requirements relevant to the development of context-aware systems.

3.2 Requirements

A large list of publications was surveyed with the purpose of documenting the *requirements* identified by the researchers studying context-aware systems. These requirements cover both functional and extra-functional aspects.

The first classification includes requirements which are directly related to the functional aspect of context-aware systems, like their ability to query context information on demand, or handle heterogeneous context representations.

The latter includes requirements characterizing extra-functional properties of context-aware systems, such as scalability and portability. Many of these extra-functional requirements are not specific to context-aware systems, but nevertheless are included in this list as they provide important insight into aspects that must be taken into consideration by the developers. Finally, it should be noted that in some of the extra-functional requirements an underlying middleware architecture is assumed.

3.2.1 Functional requirements

The functional requirements are discussed first, as they are more focused to the properties which are specific to context-awareness. This list includes eleven requirements which are listed in the following paragraphs.

3.2.1.1 Application specific context acquisition, analysis and detection

This requirement is about providing a uniform and platform-independent interface for applications to express their *need* for different context data without knowing how that data is acquired [120, 146]. A consistent and well-defined context access interface is clearly an important requirement for enabling easier development via separation of concerns.

3.2.1.2 Context-triggered action

Application software often decides what action to take based on the current context. The action could involve adapting to the new environment, notifying the user, communicating with another device to exchange information, or performing any other task. Middleware should provide the facilities for application software to define such context-triggered actions so as to transparently invoke them whenever the corresponding context conditions are valid [146].

3.2.1.3 Heterogeneity of context data

Context information can be derived from multiple sources, both directly (e.g., hardware sensors) and indirectly (e.g., synthesized context, profile information, etc). Naturally, these types of context information differ from each other in many extra-functional properties, such as accuracy and persistence [66]. It is important that successful context frameworks offer at least some support for overcoming the heterogeneity constraints which are inherent in context information by providing appropriate operators for comparing or transforming them.

3.2.1.4 Uncertainty of context information

Many factors contribute to uncertainty in context reading. For example, a context value might be completely *unknown*, *ambiguous*, *imprecise* or even *erroneous*. For example, the location of a device might be unknown when there is no information about it, ambiguous when there are multiple readings of it which do not exactly match (e.g., two individual sensors, such as a GPS receiver and a WiFi-based locator provide different approximations of the location), or imprecise when the information is available but not to the desired granularity level (e.g., the city name is available, but not the exact street in which the device resides). Finally, the reported location of the device might be completely wrong, but this would be extremely difficult even to detect.

In [66], it is argued that most approaches described in the literature fail to handle even one of these four types of uncertainty, and none of them succeeds to completely handle all.

3.2.1.5 Context histories

As context information is dynamically generated and processed, new values continuously supersede previous ones. For example, when a new reading of a GPS receiver is performed, the new location reported by it is used to replace the previous one as the current location. However, in

many cases use of historical context values is needed as well. Dey et al argue that maintaining a context storage and history is an important requirement in order to enable arbitrary context-aware applications [46].

In [66], Henricksen identifies three typical applications of context histories, dealing with both past (i.e., stored) and future (i.e., predicted or planned) context values. For example, prediction of the WiFi signal strength can be computed based on historical context values. Or, decisions about possible adaptation might be taken based on some planned tasks as reported in the user's agenda (e.g., switching to the power-save mode to save battery if the user is expected to be out-of-office for the next few hours).

Finally, context histories are important in enabling analysis of past context values for the purpose of triggering predefined events (e.g., generate an event when the temperature changes by more than 3° Celsius in less than 30 minutes).

3.2.1.6 Inference of inter-dependencies of context

While context information deals primarily with concepts such as people, devices, places and their properties [114], important information is also encoded in the *relationships* between these concepts. For instance, people have relationships with other people (e.g., "family", "friends", etc), with certain devices (e.g., "owns", "carries", "has access to use", etc) and with places (e.g., "is currently inside this building").

The importance of inter-dependencies of context was illustrated by Efstratiou et al in [49], where it was argued that knowledge of the relationships between context types is important for taking adaptation decisions. For example, ignoring the relationship between bandwidth and battery consumption would prevent the system from reducing its dependency on bandwidth as a measure for preserving battery.

3.2.1.7 Support for multiple concurrent context-aware applications

As modern mobile devices have become more powerful, they are commonly used to run multiple applications at the same time. Providing support for multiple and concurrent context-aware applications is thus a natural requirement. Instead of employing multiple instances of the same sensors/widgets/components, it would be preferable if the applications could share the context information as it is generated by a single instance of them [6].

3.2.1.8 Transparent distribution of context data

In many cases, the context information gathered and modeled in one device is required to be distributed to different devices, both nearby and remote. Furthermore, the distribution of context information should be as transparent as possible in order to ease the development task [46]. There are many reasons why this is important. From a functional perspective, in some cases the application logic itself requires context distribution. For example, a communication application might reveal the callee's status (e.g., available, busy, or unavailable), before a call is even attempted. From an extra-functional perspective, context distribution is also important as it can enable richer context information (from multiple, distributed sensors) and more efficient operation (sharing of common sensors installed only in a subset of the devices requiring the corresponding context information). For example, a higher-precision car-installed GPS receiver is preferred to a smart-phone's WiFi-based location-inferring sensor when the user is driving.

3.2.1.9 Privacy of context information

Privacy is one of the most often-cited criticisms of ubiquitous computing and may be the greatest barrier to its long-term success [72]. As the frameworks have access to sensitive user information (e.g., static information such as the user profile, and also dynamic information such

as the user's location), context-aware frameworks should strive to protect sensitive context information. This requirement is important in cases of context distribution and context storage. For instance, when context is shared over networks, it might be important that the communication is carried over a secure connection. Furthermore, when context is stored in permanent memory (e.g., on a hard disk or a *Solid State Drive* (SSD)), it is important that the data is encrypted to prevent unauthorized access in case the mobile device is lost or stolen.

3.2.1.10 Traceability and control of context-aware behavior

As context-aware systems typically employ self-adaptation logic which is quite complex, there should be some provisions for allowing the developers (and the users) to inspect the information flow—and, when possible, manipulate it—in order to provide users with adequate understanding and control of the system, and to facilitate debugging [69]. Traceability significantly improves user acceptance, as users have been found to be more tolerant of incorrect actions taken by context-aware applications if they are able to understand that they have a rational base [68, 111].

3.2.1.11 Interoperability and standards

In a setting where the context-aware applications are composed of various (possibly distributed) components, realizing both roles of providing and consuming context information, interoperability among those is an important requirement. In practice, interoperability is the requirement for some standards which guarantee that any software conforming to them can be seamlessly integrated with other compatible components. As argued by Wright in [142], the lack of open standards is a major hurdle that needs to be overcome before contextual computing reaches the masses.

3.2.2 Extra-functional requirements

As mentioned earlier, the extra-functional requirements are not specific to context-aware systems but nevertheless they discuss important aspects that must be taken into consideration. This non-exhaustive list includes fifteen items, which are ordered according to their relevance to the properties of context-aware systems.

3.2.2.1 Code reuse

By implementing the context-aware related functionality (i.e., context sensing, processing, storing, etc) outside the application, multiple applications can reuse the same component, either one at a time or simultaneously (see sub-subsection 3.2.1.7). While the requirement for component reuse was identified as early as in [46], in this thesis it is argued that code reusability can be extended even further, down to sub-components of the context sensors (see chapter 6).

3.2.2.2 Modularity

As different applications and different deployment platforms have different requirements and capabilities, a modular architecture is needed to ensure that those and only those features which are explicitly needed in each deployment setting are used. A modular architecture can enable this by allowing different variants of the middleware to be deployed on varying platforms as needed. Also, modular architectures provide the advantage of better and more resource-efficient customization to the dynamic needs of the deployed applications.

3.2.2.3 Separation of concerns

Separation of concerns is a requirement that enables a more disciplined and efficient method for developing context-aware applications [46]. For instance in [105] and in [107] the concerns

of context-awareness and self-adaptiveness are treated independently of the typical business logic of the application. As the developers are allowed to focus on individual concerns of the development of context-aware systems at each time, the overall development and maintenance tasks are rendered easier, faster, and more cost efficient.

3.2.2.4 Dynamic behavior

Many context-aware frameworks use specialized context sensing components which are used as wrappers around hardware sensors or Operating System libraries (e.g., context widgets [44] and context plug-ins [110]). Arguably, when these are deployed on mobile devices which are characterized by limited battery resources, it should be possible to dynamically activate or deactivate them as needed. This should be done according to the actual run-time needs of the deployed applications, while at the same time minimizing the resource consumption.

3.2.2.5 Resource efficiency

The requirement for resource efficiency refers to the ability of context-aware frameworks to operate on lightweight devices while imposing only a small footprint on resources such as CPU, memory and battery. This requirement is particularly important for small, mobile devices which are characterized by limited resources because of space, shape, and weight constraints.

3.2.2.6 Ease of building

This requirement refers to abstraction and information hiding [102]. The developers of context-aware applications should be exposed to just a minimal API of a provided framework, allowing them to access the functionality they need (i.e., context access and processing) without requiring them to be aware of the underlying details [46].

3.2.2.7 Platform independence

Another characteristic of mobile and ubiquitous computing is the variety of devices and platforms they are deployed on. For instance, there are currently multiple smart-phone platforms in use, along with numerous programmable embedded devices. Because of this, it is important that the proposed solution faces this problem by allowing automatic or, at least, straight-forward porting of the necessary components.

3.2.2.8 Lightweight architecture

This requirement stems from the need for both easy development of context plug-ins and constrained resources available in the target devices (which are typically smart-phones and embedded devices). However, although lightweight, the proposed architecture should be complete in terms of functionality.

3.2.2.9 Support for mobility

Most interesting context changes occur when the users are mobile because of the high variability that characterizes the environment at that time. Consequently, context-aware applications are more useful and valuable when the users are on the move, which implies that the applications should run either on mobile devices (e.g., hand-held or wearable), or they should be embedded in everyday objects (e.g., intelligent umbrellas or embedded displays) that interact with the users as per the ubiquitous computing paradigm [139, 140].

3.2.2.10 Ease of deployment and configuration

The hardware and software components required for the formation and deployment of context-aware applications must be easily deployed and configured to meet user and environmental requirements. Special attention is required to allow even non-experts to perform these tasks, as in many cases the end-users are the *consumers* (e.g., as in smart-home environments) [69].

3.2.2.11 Uniform development support

Few of the commonly used programming languages that exist today include basic support for expressing context-awareness. Even if context-aware languages exist in the future, support for expressing context awareness on a conceptual level will most likely differ across different languages. This poses a problem when developers of context-sensitive application software need to reuse their designs in different languages, hardware, or operating systems. To overcome this limitation, middleware is commonly used, as it can provide a uniform way for expressing the application's context-awareness without restricting itself to specific languages, operating systems, or environments [146].

3.2.2.12 Evolution

This requirement was first identified in [46]. The authors argue that evolution in context-aware applications can range to any of three ways: changing the context used by the applications, changing the way the context is acquired and changing the way the applications react to context. The most interesting application of evolution involves the way the context is acquired. As technology evolves and better, more accurate or more efficient mechanisms become available, context-aware applications can be improved simply by replacing their context acquisition part (assuming that a

component-oriented approach is used). This requirement is also identified in [77], where the authors request the ability for the designer to change the context capture infrastructure at run-time, without having to recompile, or even stop, the existing applications.

3.2.2.13 Scalability

As many context-aware frameworks are designed to concurrently serve multiple context consumers and context providers, scalability is needed to ensure that the capacity of the system can accommodate the demand. This requirement is highly relevant to distributed context management systems, as it is argued in [39]. In its broadest sense, however, scalability is also a critical problem in ubiquitous computing where multiple embedded devices interact with each other in an autonomous manner. Satyanarayanan identified this problem and proposed what he called *localized scalability*: as the users move away from specific computing devices, the intensity of their interactions has to fade out, otherwise both the users and the computing systems will be overwhelmed by distant interactions that are of little relevance [122]. Following suit, the coupling of context providers and context consumers should also conform with this approach—where location might refer to either physical or logical distance—while individual entities must also be able to cope with varying levels of demand.

3.2.2.14 Adoption of existing patterns and standards

To facilitate both a smoother learning curve for the developers and also to benefit by the existing codebase, it is important that the provided solution and its implementation follow and use existing patterns (such as the *Model-View-Controller* [52]) and standards whenever possible. For instance, COSMOS revisits four common patterns and puts them in use towards the development of context-aware applications (see subsection 3.3.14). Also, the *Open Service Gateway initiative*

(OSGi) is quickly becoming a standard mechanism for realizing component-based systems for mobile and embedded devices (e.g., the architecture presented in chapter 5 is based on the OSGi component framework [132]).

3.2.2.15 Fault tolerance

Sensors and other components are likely to fail in the course of ordinary operation of a context-aware system. Failures might be due to hardware problems in the underlying sensory equipment, or software problems related to bugs. In cases of distributed computing settings, communication interruptions might also occur. The context-aware systems should be able to continue operation in the event of failures, without consuming excessive resources to detect and handle failures [69].

3.3 Software support for context-awareness

While the previous section focused on the requirements identified in the literature, this section examines existing frameworks representing the state-of-the-art in the development of context-aware applications.

Over the past decade and a half, many approaches were proposed for enabling the development of context-aware applications. Some of them are more general in focus, while others are specific to context-awareness. Naturally, different approaches cover different subsets of the requirements presented in section 3.2 to various extends.

In general, software support for applications can be classified as libraries, frameworks, toolkits and infrastructures. These approaches are not mutually exclusive. As argued in [73], it is sometimes useful to have all of these. The following subsections discuss some representative approaches from the state-of-the-art, in roughly chronological order.

3.3.1 Schilit's framework

In his PhD thesis, Schilit presented a system architecture for mobile, distributed computing [124]. Parts of his work were used towards the ParcTab application [138]. That was the first architecture to explicitly support context-aware computing (mostly location oriented), and as such it simply aimed at *making it possible* to build context-aware applications [125]. For instance, the architecture does not support or provide guidelines for the acquisition of context, a task that is rather left for the developers. This fact hinders the building of context-aware applications, as the developers often have to unnecessarily re-implement significant portions of code. Furthermore, many of the functional requirements, such as support for uncertainty and histories are not supported, thus increasing the burden left to the developers.

3.3.2 Stick-e framework

The Stick-e framework was designed with the purpose of addressing the needs of context-aware notes (for example to make up a tour guide [29]). The basic idea of the stick-e framework is to allow for notes to be both automatically tagged with context metadata (such as location and nearby people) and then being presented to the users in a context-aware manner using a triggering engine [30].

Stick-e notes are attached to both concepts (such as persons, places and time, etc) and the content they represent (such as information, actions, interfaces, etc). This model is complemented with a triggering engine [30] which enables context-aware presentation of information (i.e., stick-e notes) based on a set of rules. For example, the context of *temperature*, *location*, and *nearby persons* could trigger a "go to the beach recommendation" stick-e note [103].

The application model of the Stick-e framework follows the *Model-View-Controller* design pattern [52]. The model includes the information that is to be displayed and also the view representing the system's user interface (e.g., the display). The controller is then used to mediate the operation of the model, according to the triggers defined in the triggering engine, which is eventually depicted in the view.

The main limitation of the Stick-e framework is that it is too focused. While it has been demonstrated in a number of context-aware applications, it uses a rather monolithic context model and is limited to rule-based decisions, making it unsuitable for arbitrary types of context-aware applications.

3.3.3 CyberDesk

The CyberDesk is a framework designed for providing self-integrating context-aware services [47, 141]. This framework is primarily concerned with what its authors term as *virtual context*, which includes the email addresses of people relevant to the application, dates, their names, etc.

The CyberDesk was one of the first frameworks to leverage separation of concerns, in terms of enabling the applications to specify the context types they are interested in, and let the framework handle its acquisition, processing and delivery. Nevertheless, this framework is rather limited as it focuses primarily at only one type of context: user's current selection of text. Furthermore, it has additional limitations such as missing support for multiple simultaneous applications and missing support for context querying or storing. The latter were the basic reasons the authors deprecated this architecture in favor of the Context Toolkit (see subsection 3.3.4), as it is reported in [46].

3.3.4 Context Toolkit

When it was first introduced the Context Toolkit [44] was a pioneering approach, and even today it is still widely referenced in the literature. The basic idea of this approach was to simplify the development of context-aware applications by enabling the reuse of specialized components, in a similar way to how *widgets* facilitate the development of advanced GUIs.[1]

The proposed abstractions include: widgets, aggregators, interpreters, services and discoverers. A widget is a component that is responsible for acquiring context information directly from a sensor. The aggregators can be thought of as meta-widgets, taking on all capabilities of widgets. They also provide the ability to aggregate context information of real world entities such as users or places and act as a gateway between applications and widgets. Interpreters transform low-level information into higher-level information that is more useful to applications. Services are used by context-aware applications to invoke actions using actuators. Finally, discoverers are used to locate suitable widgets, interpreters, aggregators and services.

The toolkit is implemented as a set of Java APIs that represent its abstractions and use the HTTP protocol for communication and XML as the language model. Components implemented in other languages can also interoperate via the use of Web standards. The authors of the Context Toolkit argue that by using its abstractions for the development of context-aware applications (rather than following an ad-hoc approach), the developers benefit in terms of *ease of building*, *support for reuse* and *support for evolution* [46]. Finally, an extension to the Context Toolkit was proposed by Newberger and Dey for providing user control of context-aware systems using an approach called end-user programming [97].

[1] *Graphical User Interface*

3.3.5 Reconfigurable Context-Sensitive middleware

The *Reconfigurable Context-Sensitive Middleware* (RCSM) [146] is designed to facilitate applications that require context-awareness and spontaneous, ad-hoc communication. The authors propose a model that applies triggering at the object level. RCSM proposes an architecture which splits applications to a context-sensitive interface (consisting of a set of triggers that associate context events to actions) and a context independent implementation (realizing the appropriate actions).

While it incorporates some interesting ideas, this approach employs a rather simplistic context model that is based on simple variables. Furthermore, functional requirements such as uncertainty, relationships, and historical context information are not addressed, resulting in a significantly constrained framework.

3.3.6 Context-Oriented Programming

The authors of [71] propose a language extension for creating context-aware applications using so-called *Context-Oriented Programming* (COP). COP concentrates on providing support for developing the context-aware behavior of an application. To enable this, the authors introduce the concept of *layers of behavior variations* which are dynamically selected using *multi-dimensional message dispatch*.

In COP, low-level context sensing and reasoning are neglected by stating that simple object orientation is sufficient for context modeling. However, this is a rather unjustified simplification, as it bypasses important requirements including interoperability, extensibility and semantic-based separation of concerns.

3.3.7 Context Information Service

The *Context Information Service* (CIS) [78], developed by and used in Aura [129], provides a database approach for context-aware applications where queries are encoded in a SQL-like language. In this approach, the context information is collected on demand from distributed infrastructures (i.e., context providers).

The authors identify three requirements that they aim to satisfy with CIS: allow for *synthesis* of the required context information, facilitate *efficient* information providers, and support *dynamic attributes*. Their architecture follows an approach where a complex query is first received and analyzed by the CIS, and then broken down to more elementary context queries. The latter are forwarded to the corresponding context providers, and their replies are collected and synthesized into the result of the original query which is eventually communicated back to the original client.

While the proposed query language is SQL-like, it does not require an actual database, making it significantly lighter and suitable for mobile devices. Furthermore, the general definition of context providers allows for both static ones (e.g., a phone database) and dynamic ones (e.g., temperature). The use of a predefined set of context attributes (i.e., accuracy, confidence, last update time, and sample interval) allows for rich queries, which as a result shift the burden of computation from the client to CIS.

The CIS framework is efficient, partly because of its support for caching at several points in the architecture, and scalable as its on-demand approach minimizes processing to the absolutely necessary (i.e., query only when needed and to the granularity that is needed). Arguably, one of the advantages of this approach is that it delegates some of the power of traditional *DataBase Management Systems* (DBMS) to context clients, via a SQL-like query language. CIS, however, does not deal with privacy and security concerns as they are beyond its scope [78]. Furthermore,

despite its heavy reliance on distributed context providers, the authors do not comment on the ability of CIS to overcome even simple faults, such as network disconnection.

3.3.8 Context-aware reflective middleware system for mobile applications

Another approach for aggregating context has been proposed by Capra et al in [33]. The *Context-aware reflective middleware system for mobile applications* (CARISMA) approach uses reflection and metadata to build a system that supports context aware applications. Applications pass metadata to the middleware. These metadata constitute a policy as to how the applications want the middleware to behave as a result of a specific context occurrence. As context and application needs continuously change, one cannot assume that the metadata are static. Therefore, applications also use the reflection mechanisms offered by the middleware to inspect their own metadata, and possibly alter them according to the changing needs. The metadata format is standardized using XML Schemas.

This framework provides the developers with an abstract syntax for defining application profiles. These profiles are passed down to the middleware, and the applications can change them during execution. The profiles are used by the middleware to realize the application context-aware behavior. This is achieved in terms of defining context-dependent adaptation situations, determined through utility functions [32].

CARISMA's main contribution is a microeconomic-inspired mechanism for enabling context-aware adaptation. This mechanism is based on utility functions—similar to those discussed in subsection 2.3.3—and is capable of resolving conflicting decisions [34]. However, this mechanism does not deal with the mechanisms required for low-level context management. Also, it does not deal with many of the requirements which are related to varying context providers like heterogeneity, uncertainty and dynamic behavior.

3.3.9 Context Fusion Networks

The *Context Fusion Networks* (CFN) approach is proposed by Chen et al [39]. It allows context-aware applications to select distributed data sources and compose them with customized data-fusion operators into a directed acyclic information fusion graph. Such a graph represents how an application computes high-level understandings of its execution context from low-level sensory data. The CFN aims to address four functional challenges: *flexibility*, *scalability*, *mobility* and *self-management*.

CFN was prototyped in an implementation named *Solar* [38], a flexible, scalable, mobility-aware, and self-managed system for heterogeneous and volatile ubiquitous computing environments. Solar consists of a set of functionally equivalent nodes, coded *planets*, which peered together form a service overlay. This service realizes a *Distributed Hash Table* (DHT), based on P2P routing protocols. Planets are defined as execution environments for operators to cooperatively provide operator-management functionality, such as naming and discovery, routing the sensory data through operators to applications, operator monitoring and recovery in face of host failures, and garbage collecting operators that are no longer in use.

While CFN satisfies many of the requirements identified earlier in this chapter, its architecture lacks modularity, making it difficult to provide platform-independent implementations. Furthermore, its reusability is limited to specialized operators, rather than higher-level components. Finally, as CFN is highly oriented towards distributed context fusion, it is makes it difficult to realize light implementation targeting centralized deployments only.

3.3.10 Pervasive Autonomic Context-aware Environments

The *Pervasive Autonomic Context-aware Environments* (PACE) [69] is a middleware-based approach. It consists of a set of components and tools that have been developed according to three

design principles: First, the model of context information used in a context-aware system should be explicitly represented within the system. Second, the context-aware behavior of applications should be determined, at least in part, by external specifications that can be customized by users and evolved along with the context. Finally, the communication between application components and between the components and middleware services should not be tightly bound to the application logic.

This middleware is based on a *context management system* which provides aggregation and storage of context information and performs query evaluation. Furthermore, it contains a distributed set of context repositories where each repository manages a collection of context models. A *preference management system* is also provided to facilitate decision-support for context-aware applications. The actions preferred by the user are determined by the evaluation of preferences with respect to application state variables and context information stored by the context management system. To facilitate interaction between application components and the context and preference management systems, a *programming toolkit* is defined and implemented in Java, using its *Remote Method Invocation* (RMI) for communication. Additional development tools are also provided to assist with the generation of components and with the development and deployment of context-aware systems, starting from context models specified in the context management system.

The PACE middleware provides support for heterogeneity, mobility, traceability and control of context flow, and ease of deployment and configuration. Requirements like scalable deployment and code reuse (e.g., of context sensors) have not been addressed.

3.3.11 Context Modeling Language

Henricksen and Indulska developed a graphical modeling approach, the *Context Modeling Language* (CML), as a tool to assist designers with the task of exploring and specifying the context requirements of a context-aware application [67, 68]. This model defines constructs for *types of context* (in terms of *fact types*), their classification (i.e., whether it concerns *sensed*, *static*, *profiled* or *derived* context), associated *quality metadata* and *dependencies* amongst different types of context.

The authors of CML selected to extend the modeling concepts of the *Object-Role Modeling* (ORM) [64] because of its superior formality and expressiveness compared to, for instance, the *Entity-Relationship* (ER) model, and also because of the presence of a mapping to the relational model allowing for straightforward representation in relational databases.

Two programming models were examined: First, the *branching* model facilitates a context-dependent choice among a predefined set of alternatives. Similar to using utility functions [135], the branching approach specifies a function that given a set of choices and context state, it produces a mapping of choices to scores (it could be a finite set of scores or simply a value in the range from 0 to 1). On the other hand, the triggering model supports an asynchronous style of programming, where certain context changes trigger different actions. The model proposed by the authors, similar to other triggering approaches [30], allows for binding certain transitions of context *situation abstractions* (or even sequences of certain transitions) to predetermined actions. This model also associates triggers with a lifetime (for instance, the action is triggered "once", or "n times", or "until $< end >$", etc).

The authors evaluate the CML in terms of three software quality metrics: *code complexity* (in Lines of Code or LOC), *maintainability and support for evolution*, and *reusability*. The authors show that only a small fraction of the overall code implementing a case study application concerns the context-aware logic. Furthermore, they argue that maintainability and evolution are appropriately supported, as the loose coupling of the source code from the underlying context model makes it easy to change the latter in response to changing requirements. Finally, reusability was demonstrated in terms of building subsequent applications that reused part of the existing models. However, while the CML-based context model is quite powerful, it is also significantly complex, making its adoption a particularly difficult task for developers. Although the generated source code is simple, learning all the concepts and semantics of the CML is a complex task, with a steep learning curve, preventing it from being widely adopted. Finally, the authors do not consider distribution of context in their requirements, and they do not discuss how it could be enabled using CML.

3.3.12 Mobility and adaptation enabling middleware

The *Mobility and adaptation enabling middleware* (MADAM) architecture [50, 54], building partly on top of the *Framework for adaptive mobile and ubiquitous services* (FAMOUS) approach [63], follows an architecture-centric approach where architectural models are represented at runtime to allow generic middleware components to reason about and control adaptation in a reflective manner. In this approach, context is assumed to be provided in a raw form from simple context sensors, employing a simplistic context model. Context-awareness is mainly enabled at a higher level, where adaptation reasoning enables scoring alternative architecture instances, selecting the one that maximizes the end-user utility [18, 19, 20].

This approach differs from most context-aware enabling frameworks, as it does not employ a sophisticated context-awareness model. Rather, context collection and management are considered at a different layer, leaving the responsibility of reacting to context changes to a sophisticated, utility-function based model. While the underlying context model and management mechanism are simplistic, barely satisfying many functional requirements (including heterogeneity, uncertainty, inference of interdependencies of context, traceability and control), it does show some interesting and novel properties, such as support for dynamic behavior and evolution. It should be noted that this architecture is the predecessor of the pluggable context middleware described in this thesis.

3.3.13 EgoSpaces

EgoSpaces [79] is a framework for facilitating rapid development of context-aware applications, realizing a *tuple space*-based middleware (see subsection 3.4.2). This approach focuses on mobility and, in particular, on ad-hoc network topologies which are both dynamic and unpredictable. The authors propose an application-level approach to context, where context is generalized so that different types of it can be dealt with in a similar manner. Similar to other frameworks, this approach aims at providing high-level abstractions for easing the programming burden.

The EgoSpaces framework puts forward the concepts of transient tuple space sharing, flexible tuple representation, and declarative view specification. Views are sets of tuples that satisfy some data constraints and are owned by agents that satisfy the agent constraints. Views reside on hosts, which satisfy the host constraints, and these hosts must lie within the boundaries defined by the network constraints.

As this approach is highly customized to the context distribution aspect, its proposed tuple-based model (layered in *computational*, *data representation*, and *view concept*) fails to explicitly

address requirements such as uncertainty, inference of interdependence of context, and traceability and control.

3.3.14 Context entities composition and sharing

Context entities composition and sharing (COSMOS) [41, 117] is a component-based framework for managing context information in ubiquitous context-aware applications. This framework decomposes context observation policies into fine-grained units called *context nodes*, and combines *software components* [131] and *design patterns* [52] to define the foundation of its architecture. At its core, COSMOS proposes a three-step process where first context is *collected*, then *interpreted*, and finally used for *situation identification*.

Context management policies are expressed as hierarchies of context nodes using a dedicated composition language. This hierarchy consists of leaf-nodes (corresponding to sensors collecting context from external sources) and other nodes (such as averaging and translator operators, data mergers, inference operators, and threshold operators).

The authors of COSMOS propose the reuse of four popular design patterns in order to support mapping of the context policies to the context nodes hierarchy: the *Factory*, *Composite*, *Flyweight* and *Singleton* patterns [52]. These four design patterns reflect variation points in the architecture, which are then configured into COSMOS descriptions.

The use of components and design patterns allows for dynamic reconfiguration and evolution (see sub-subsection 3.2.2.12) once the policies have been specified and deployed. Similar to other context frameworks, such as the context toolkit (see subsection 3.3.4), COSMOS aims to simplify the development of the context-aware logic of applications by hiding the reasoning logic in abstract components, namely *context nodes* (or *widgets* in the Context Toolkit).

Not enough information is available concerning the framework's underlying context model. While some extra-functional requirements are addressed by COSMOS (such as separation of concerns and evolution), several functional ones (see subsection 3.2.1) are not, including heterogeneity, uncertainty, transparent distribution of context, etc.

3.4 Other approaches

Besides pure context-aware frameworks, other approaches have been proposed in the literature for enabling *intelligent mobile applications*. Many of these focus on the communication aspect, which is particularly important (and challenging) in the context of highly varying environments. The following subsections briefly describe the *agent* and *tuple-space* based approaches.

3.4.1 Agents

The Adaptive Agent Architecture [85] and Hive [94] are both agent-based middleware for supporting multi-modal applications in ubiquitous computing environments. Agents are abstractions used to represent sensors and services. In the Adaptive Agent Architecture, when a sensor has data available, the agent representing it places the data in a centralized blackboard. When an application needs to handle some user input, the agent representing it translates the input and places the results on the blackboard.

Applications indicate what information they can use through their agents. When useful data appear on the blackboard, the relevant applications' agents are notified and these agents pass the data to their respective applications. The blackboard provides a level of indirection between the applications and sensors, effectively hiding the details of the sensors. Similarly, Hive and MetaGlue use mobile agents to acquire information from the environment and distribute it to applications. The agent-based approach is suitable for implementing a context-aware framework,

but a number of concepts would have to be added, including the ability to interpret, aggregate, and store context information [46].

3.4.2 Tuple spaces

Limbo [42], JavaSpaces [51], and T-spaces [144] are all tuple spaces-based infrastructures based on the coordination paradigm [57] and the original Linda tuple space [56]. Tuple spaces offer a centralized or distributed blackboard into which items—termed tuples—can be placed and retrieved. A representative example of using tuple-spaces in building context-aware systems is the EgoSpaces which was discussed earlier in subsection 3.3.13.

Tuple spaces provide an abstraction layer between components that place tuples and components that retrieve tuples. Although this abstraction meets the separation of concerns requirement, it does not provide support for interpretation, aggregation, or persistent context storage, essential requirements for context-aware computing [46].

3.5 Hardware support for context-awareness

Since Weiser's vision of ubiquitous computing almost two decades ago [139, 140], the area of context-aware computing has undergone extensive research, generating important results, experiences and insights for developers of context-aware applications. Despite these long-term efforts however, until recently only very few commercial applications and products were successful in incorporating context-aware behavior. Some of those just exhibit simple context presentation properties. For example, the ambient umbrella [1] is designed to glow in different colors based on the local weather forecast, indicating to their owners whether they should take them along when exiting their home. Besides this simple example, however, very few applications have been commercially successful in showcasing context-aware behaviors, especially complex ones.

Nevertheless, this seems to be changing now. More and more applications start offering context-aware properties, building on the success of new smart-phone technology such as Apple's iPhone [2], NOKIA's S60-based smart-phones [10] and an increasing number of devices based on the *Open Handset Alliance's* Android platform [11]. One common characteristics of these modern devices is that they come equipped with numerous sensors [143]. These sensors provide context types such as location (via GPS assisted and WiFi look-up-based sensors), acceleration, proximity, ambient light and noise conditions and even direction (via digital compass sensors). Also, with their increasingly more powerful communication capabilities, these newly introduced smart-phones are quickly changing the topology of context-aware applications.

Already, more and more production-status applications appear in the headlines thanks to their context-aware behavior. For example, *Enkin* [3] and Google's *Sky map* [4] both provide a virtual view-port through which users view information annotations attached to real objects in their mobile device's display. In Enkin, the device's camera is used to display the real-world image pointed by it. At the same time, context information from the attached location, orientation and compass sensors is utilized to determine which objects are in the display so that visual annotations are attached to them (e.g., a metro station's name and the distance to it). Similar to Enkin, Sky map provides a virtual view of the night sky, with name annotations to planets and stars. As the users point their device in the sky, context information regarding the device's location, orientation and direction is used to update the display accordingly. A more widely used application is Google's Latitude which allows its users to view the location of their friends and family in real-time. This kind of applications open up new horizons of possibilities but—undeniably—also introduce serious privacy concerns.

3.6 Summary and conclusions

Context-awareness has been actively researched by the software engineering community for almost two decades. However, only recently are context-aware applications becoming popular, partly because of the quick proliferation of powerful smart-phone devices and the nearly ubiquitous network access in populated areas. This chapter presented a representative subset of the state-of-the-art from the perspective of designing and building context-aware systems. In this respect, the examined approaches were analyzed in three aspects: *challenges*, *requirements* and *technical approaches*. Combined, these provide a broad view of the current results in this area, as well as the main challenges that have motivated them and the requirements that have shaped them.

Table 1: Summary of requirements for context-aware applications

Functional requirements	**Extra-functional requirements**
Application-specific context acquisition, analysis and detection	Code reuse
Context-triggered action	Modularity
Heterogeneity of context data	Separation of concerns
Uncertainty of context information	Dynamic behavior
Context histories	Resource efficiency
Inference of inter-dependencies of context	Ease of building
Support for multiple concurrent context-aware applications	Platform independence
Transparent distribution of context data	Lightweight architecture
Privacy of context information	Support for mobility
Traceability and control of context-aware behavior	Ease of deployment and configuration
Interoperability and standards	Uniform development support
	Evolution
	Scalability
	Adoption of existing patterns and standards
	Fault tolerance

The identified requirements are summarized in table 1. This table serves as a quick reference to the requirements that have driven the research directions in general context-awareness. Furthermore, these requirements were adopted by the author of this thesis while designing and realizing the development approach and middleware architecture presented in chapters 4, 5 and 6.

As the process that was followed to gather these requirements was based on studying and analyzing the current state-of-the-art, the resulting list of requirements is quite extensive and with a rather broad scope. In result, many of these requirements, especially the non-functional ones, are applicable in general software engineering. This is the reason these requirements were roughly ordered according to their significance and relevance to context-aware systems, as it was mentioned in subsection 3.2.2.

Previous research focused on proposing frameworks that facilitate the development and deployment of context-aware applications. These were designed with one fundamental goal: shifting the complexity of the context-aware logic from the application onto the middleware, thereby simplifying their construction and maintenance. However, existing solutions satisfy only a subset of the detected requirements (see section 3.3) while some of them exhibit other shortcomings too. For example, some solutions are specific to rather narrow domains (see the EgoSpaces framework in subsection 3.3.13), or they define simplistic context models which render them impractical for complex applications (see Schilit's approach and RCSM in subsections 3.3.1 and 3.3.5).

These observations have motivated the research presented in this thesis. In order to address the identified challenges, while at the same time satisfying the requirements enumerated in the literature, a development methodology and middleware architecture are proposed. The methodology is presented in chapter 4 and the middleware architecture in chapter 5. Chapter 6 proposes a model-driven development approach for building parts of the applications, and finally chapter 7 evaluates the proposed solution and shows how it enables software developers to produce context-aware systems faster and more efficiently.

Chapter 4

A development methodology for creating context-aware applications with separation of concerns

This chapter presents a novel approach for creating context-aware applications in a flexible and cost-efficient manner. This is achieved by separating the concern of context production from that of context consumption and also by defining specialized components that can be reused to form the context-sensing and context reasoning logic of the applications. This approach has the following advantages. First, it facilitates software reusability by specifying context providers as reusable components. Also, it allows the developers to collect and model context information without having to worry about the underlying mechanisms used to collect and encode it. Additionally, it provides them with a powerful context access mechanism which enables them to synchronously query context data which satisfy certain conditions, or register for asynchronous notifications by specifying conditions that must be met before they are notified.

The chapter starts with an introduction to how the problem is modeled in terms of context spaces, and then it proceeds by presenting how separation of concerns can be utilized to allow for independent context providers and context consumers. This is followed by the description

of the proposed development methodology, which guides the developers to analyze the context-aware aspects of their applications and to reuse existing software components to realize them. The chapter concludes with descriptions of the context model used to encode context information and a context query language which can be used to access it.

4.1 Introduction

In order to contextualize the proposed development methodology, a requirements-driven approach is followed. At this point, only a subset of the functional requirements identified in chapter 3 are considered. A detailed evaluation of the methodology described in this chapter and the accompanying middleware architecture described in chapter 5, in relation to the requirements summarized in table 1, is presented in chapter 7.

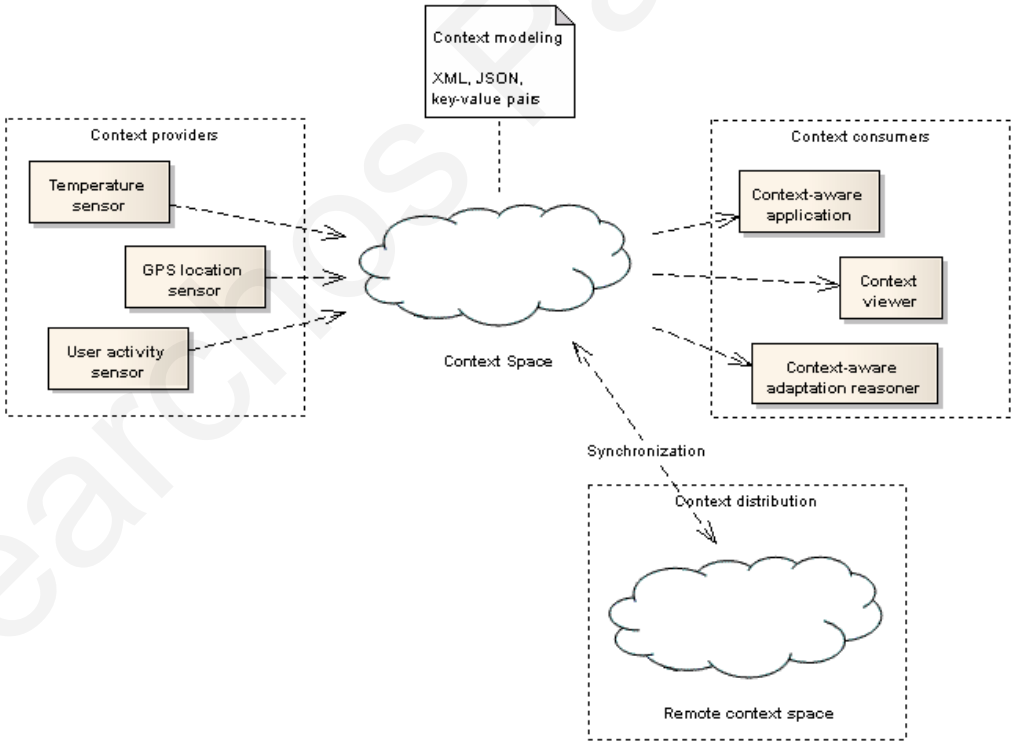

Figure 7: Using context spaces to illustrate the requirements for the developed framework

The main goal of this development methodology is to provide a flexible mechanism for accessing context information, including both low-level sensory data as well as high-level, derived context concepts. This mechanism should facilitate the development on both context providers and context consumers. The former are specialized components which produce context information either by interfacing with underlying hardware sensors and operating system (OS) libraries, or infer higher-level context concepts by processing lower-level context information. The latter are typical context clients like context-aware applications, context monitoring tools or middleware components utilizing context information. Furthermore, this mechanism should accommodate such concepts such as context distribution and domain-specific query languages. An abstraction of this model is illustrated in figure 7.

Besides general extra-functional requirements like ease of development, platform independence and lightweight implementation, the design of the proposed methodology is primarily focused on the functional requirements of:

- enabling software reuse via separation of concerns (i.e., separating the concerns of developing the context providers from that of developing the context consumers);
- allowing dynamically added—and removed—components (both context providers and context consumers);
- enabling distribution of context data;
- facilitating interoperability among context providers and context consumers, regardless of whether these are locally or remotely deployed, seamlessly overcoming heterogeneity constraints;
- providing a powerful context access interface, facilitating rich context queries.

The first requirement refers to enabling component reusability. For instance, it allows the developers to browse well-known repositories and select appropriate components in an off-the-shelf manner [65, 131] for realizing parts of their context-aware applications. A dynamic architecture enables better exploitation of the deployed infrastructure, while also contributing to better fault tolerance. For example, in many situations mobility may result in environments where new (or richer) context information becomes available as the user moves within range of various, distributed context providers. For instance, consider a scenario where an application utilizes higher-precision location data provided by a car-installed GPS receiver when inside the car, as opposed to lower-fidelity location information provided by a WiFi-based location sensor installed in the smart-phone. This scenario also highlights the importance of allowing context distribution, where context information from one device can be seamlessly shared with applications deployed on another device. Interoperability is also an important requirement for highly reusable components. It is thus important to follow a simple modeling and encoding approach, adopting existing standards as much as possible. At the same time, this approach should also facilitate automated mechanisms for overcoming heterogeneity-related constraints. Finally, rich context query languages are also desired, as they can facilitate the development of applications where the mechanisms realizing their context-aware logic are kept outside the application and inside the middleware, as much as possible.

4.2 Developing with separation of concerns

This section introduces a development methodology aiming to simplify the production of context-aware applications by allowing the developers to leverage the principle of separation of concerns. For this purpose, an application model is first presented, splitting the application's

business (i.e., functional) logic from its context-aware behavior. The former is independently designed and implemented as usual. Its context-aware (i.e., extra-functional) logic is then realized via the development or reuse of appropriate context sensors and reasoners. These sensors and reasoners are developed as independent components and are deployed on a middleware architecture which facilitates interfacing them with context consumers. The business logic is interfaced to the context-aware logic by registering for—and receiving—high-level or low-level values of specified context types, either synchronously or asynchronously. Technically, the binding of these components is done at run-time and is controlled by the middleware in a way which aims at optimizing the resource consumption.

4.2.1 Application model

This thesis adopts an application model which is inspired from the literature. By definition, ubiquitous computing is quite broad [139, 140] and subject to various interpretations. In [25], Banavar et al identify three aspects which can arguably encompass pervasive computing:

- a *device* is a portal into an application/data space, *not* a repository of custom software managed by the user;
- an *application* is a means by which a user performs a task, *not* a piece of software that is written to exploit the device's capabilities;and
- the *computing environment* is the user's information-enhanced physical environment, *not* a virtual space that exists to store and run software.

This thesis adopts this model. As part of context-aware adaptation, a user is assumed to utilize multiple devices, both mobile carry-ons and embedded in the environment. The user interface is

not a fixed functionality provided by inflexible code, but rather can adapt in runtime (e.g., switching from visual to audible UI and vice versa). The applications are also not viewed as monolithic blocks of code, but rather dynamic compositions of components and services, which are continuously adapted to better serve the functionality requested by the user (also refer to the multidimensional utility model discussed in chapter 2 and in [106]). Finally, the physical environment is viewed as an information-enhanced environment which can be used both for *sensing* contextual information and for *affecting* it towards improving its overall utility offered to the user.

To model the applications the way they are envisioned, their life-cycle must be considered. This thesis is mainly concerned with two aspects of the application life-cycle: *design-time* and *run-time*. As it was illustrated earlier (see figure 7), a context-space is used as a reference for the envisioned applications. In this model, the context space identifies the main components which comprise the applications, and which affect both their design-time and run-time life-cycle.

4.2.1.1 Design-time

During the design phase, the developers identify a set of components which are needed for realizing the application. As it will be discussed in subsection 4.2.2, a separation is introduced between the components providing the business logic (i.e., functional behavior) of the application, and those providing its context-aware behavior (i.e., extra-functional behavior). No special requirements are given for the model used within the business logic of the application, keeping the proposed methodology as general as possible. In principle, however, it is envisioned that the business logic itself is also realized through a dynamic composition of components and services. Nevertheless, the detailed description of such model is beyond the scope of this thesis. The interested readers are rather referred to [6, 54, 105, 116].

4.2.1.2 Run-time

During the run-time phase, the applications are controlled by the middleware, which automatically discovers the required components and services, and binds them accordingly. Just like in the design-time, the exact composition of the applications in run-time is assumed to consist of both components and services that are dynamically composed (and recomposed), as it is discussed for example in [118]. However, this thesis is primarily interested in the composition of the extra-functional components of the applications (i.e., those providing the context-aware behavior) with their business logic. As it will be shown in the following sections, this binding is dynamically planned based on the provided and required context types. Specialized algorithms can even be used to optimize the resource consumption by binding the most appropriate components at each time, as it will be shown in sections 5.2 and 5.7.

The rest of this chapter further discusses the development-time and the run-time life-cycle phases of the applications, providing details on both their mechanisms and their underlying context model.

4.2.2 Context providers and context consumers

As stated earlier, the proposed methodology assumes that applications are developed using component orientation. Individual components realize either one or both of two roles: *context provider* and/or *context consumer*.

Context providers are specialized components which generate context information. Usually, these components realize *context sensors*, providing context data such as *network cost*, *location*, the *user agenda*, etc. From a practical point of view, the context sensors can be realized either as software wrappers around *Operating System* (OS) libraries, or as hardware sensors (such as the "Network sensor" and the "GPS sensor"). Alternatively, they can be also realized as advanced

software components producing context data by acting as clients to appropriate services (such as an “Agenda sensor” acting as a client of an online calendar service).

The context consumers, on the other hand, are applications which realize common business logic but also require input for specific context types in order to adapt their functional or extra-functional behavior accordingly (for example, an email application which selects its synchronization mode according to the sensed *network cost*). These components or applications are also referred to as *context clients*.

Furthermore, some components realize both the context provider and context consumer roles. The *context reasoners*, for instance, are specialized context providers which take as input lower-level context data and generate higher-level context information. Consider, for example, a context reasoner which takes as input the *location* and the *agenda* of a user and combines them to infer the *user state* (e.g., whether they currently “drive”, “attend a lecture”, “sleep”, etc).

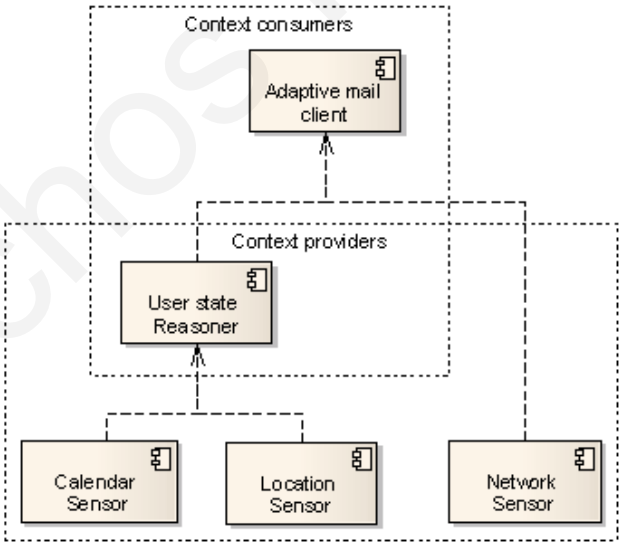

Figure 8: Context providers, context consumers and their inter-dependencies

An example illustrating the classification of context providers and context consumers is shown in figure 8. The lower-most three components (i.e., the “Calendar sensor”, the “Location sensor”

and the "Network sensor") are pure context providers. On the other hand, the upper-most component (i.e., the Adaptive mail client) is a pure context consumer. The remaining component (i.e., the "User state reasoner") is a context reasoner, realizing both roles.

In a typical deployment, the first three components would be used to provide their corresponding context data to either the "User state reasoner" or the Adaptive mail client application accordingly. The "User state reasoner" would use that input to infer higher-level information about the user's state. Eventually, the Adaptive mail client would receive input from both the context sensors and the context reasoner in order to realize its context-aware behavior. The mechanism used to provide this interface between context providers and context consumers is presented in the next subsection.

4.2.3 Interfacing with the context system

Enabling systems which are formed by loosely coupled context providers and context consumers requires some mechanism for interfacing them. One option would be to bind the context providers directly to the context consumers. However, it is argued that this solution has at least two significant disadvantages:

- Commonly shared functionality, like storing past context values and accessing them via specialized context queries, would have to be reinvented and implemented in every individual application. The same holds true for model-specific functionality, such as *Inter-Representation Operations* (IRO), i.e., transforming context information across various possible representation structures (see subsection 4.4.3).

- It would be impossible to coordinate the resource usage of the context providers, unless a common, central component was used. For example, when different applications are

both depended on the same context provider (e.g., the “Location sensor”) but have different requirements for it (e.g., activate versus deactivate the GPS sensor), various conflicts occur that would be very difficult to resolve without a centralized authority.

For these reasons, an architecture is proposed in which the context providers and the context consumers are loosely coupled through a middleware layer. A realization of such middleware is illustrated in figure 9.

Using this middleware-based architecture, the individual components (both context providers and context consumers) are only indirectly connected to each other, despite their context dependencies (which are illustrated by the horizontal arrows). These context dependencies are explicitly defined as metadata in the corresponding components and are registered with the middleware during installation. The metadata are used by the middleware to automatically resolve and activate them as it is explained later in chapter 5.

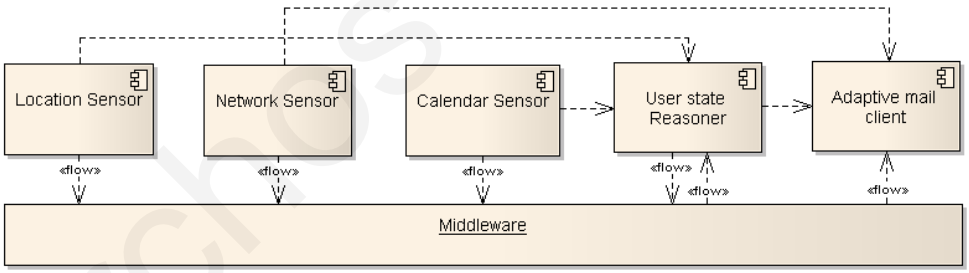

Figure 9: Middleware-based mediation of context providers and context consumers

The advantages of this middleware-based approach are as follows:

- The middleware layer allows for the centralization of commonly used functionality, such as storing in context histories, querying and distribution.

- By providing a common point of reference, the middleware can facilitate both semantic and representational interoperability among various, possibly heterogeneous, context providers and context consumers.

- Applications simply requiring the use of common context types (such as *memory usage*), often do not need to bundle any sensing code themselves since the middleware typically provides it by default.

- Multiple concurrent applications (i.e., context consumers) can share the same context sensing and/or context reasoning mechanisms, as they are only loosely coupled with the corresponding context providers.

- Better resource management can be enabled as the middleware is in a position to be aware of all the context requirements and offerings, thus enabling it to control the activation and deactivation of the corresponding context providers as needed.

4.2.4 Context-awareness and adaptation reasoning

As it was discussed in subsection 2.3.4, context data can be used in realizing both the functional and extra-functional behavior of an application. Especially in the case of self-adaptive applications, the main purpose of context data is to enable adaptation reasoning and implementation in an autonomous manner (see subsections 2.3.2 and 2.3.3). For instance, the context data is used as parameter values in specialized utility functions which are designed to compute the *fitness* of a particular application variant to the context [106].

Given the middleware-based approach discussed in the previous subsections, it is clear that in this approach the adaptation decisions are formulated bottom-up. In particular, the lowest level, elementary context data is collected first by context sensors. Those data are then further processed

by context reasoners to derive higher-level context information. Eventually, that information is used in the utility functions which are responsible to rank the possible variants, and thus enable adaptation decisions.

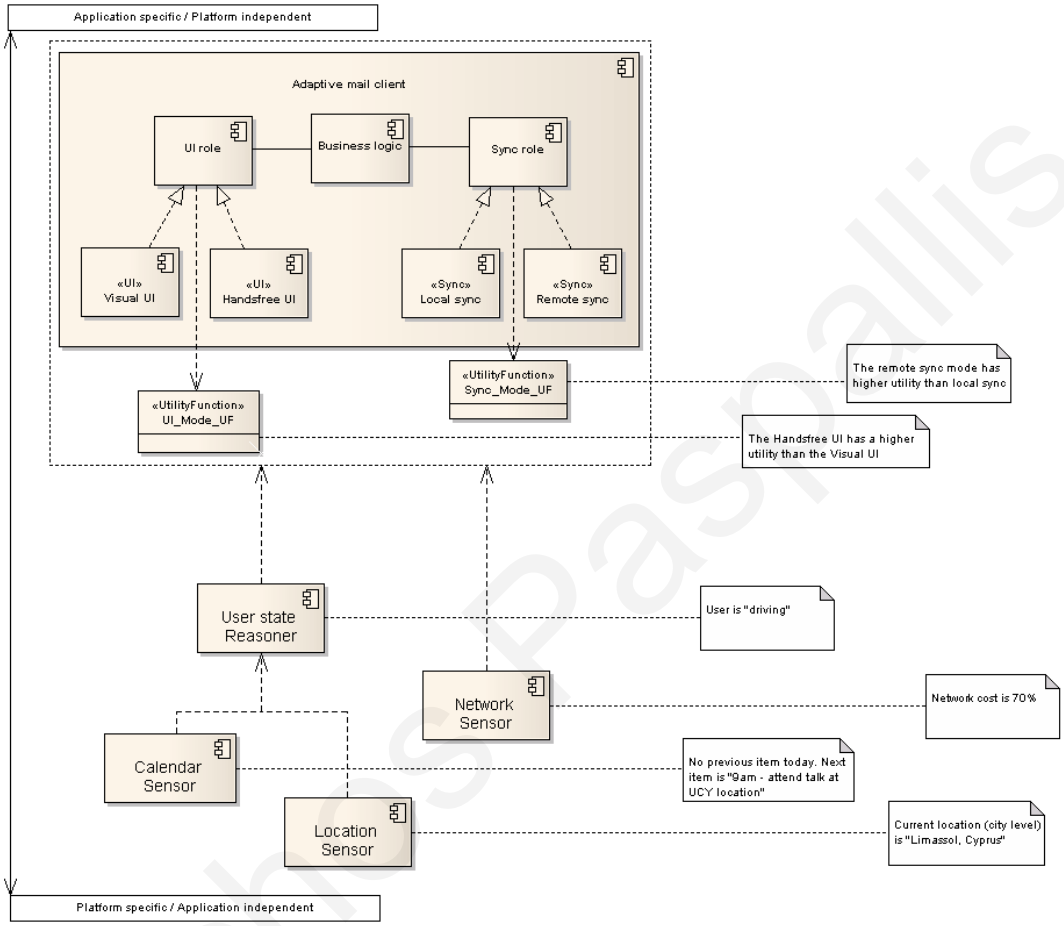

Figure 10: Hierarchical decision chain for context-aware, self-adaptive software systems

This hierarchical approach for self-adaptation decisions is illustrated in figure 10, where the bottom-most side corresponds to lower-level context data and the top-most side corresponds to higher-level context information (note that in this figure, dependencies—illustrated by dashed arrows—are *logical* rather than *physical*). Naturally, the higher an artifact appears in this hierarchy, the more significant its impact is on the adaptation decisions.

Besides being responsible for lower-level collection of context data, the context sensors (e.g., the "Calendar sensor", the "Location sensor" and the "Network sensor" shown in figure 10) are also characterized by their platform specific and application independent nature. For instance, a "GPS sensor" is typically dependent on the actual hardware sensor, or an OS library which is used to access the data, making it more platform-dependent. On the other hand, as the corresponding context type (e.g., *location coordinates*) is quite elementary, these sensors are very likely relevant across a wide range of applications, which renders them highly reusable. For this reason, it is argued that the context sensors are highly *platform-specific* and *application independent*.

Going one step up, context reasoners (such as the "User state reasoner") are generally used to process elementary context data in order to generate higher level context information. Arguably, context-reasoners are more platform independent but also highly application specific. For example, the context reasoner that computes the *user state* based on more elementary context data (e.g., the *GPS coordinates* and the *user's agenda*) can be assumed to be highly *platform independent*. This is justified as follows: the actual reasoning computations performed in the reasoner are not dependent on any specific hardware or OS libraries, but rather on other context types only. These dependencies are typically resolved via context sensors which provide the corresponding context types. On the other hand, as the context types provided by the context reasoners are more specialized, the latter are more likely to be relevant to a small subset of applications only, and thus could be considered as primarily—but not exclusively—*application-dependent*.

Finally, at the top-most level of the hierarchy, utility functions are used to compute the *fitness* of individual components to predefined roles. For example, in this figure the "UI role" is realized by two components offering *visual* and *hands-free* user interaction, and the "Sync role" is realized by two components offering *local* and *remote* synchronization options.[1]

[1]The composition of adaptive applications in terms of roles and realizations illustrated in this figure is a simplified version of existing approaches. For a detailed description of these composition approaches see [50, 54].

Clearly, the computations performed in the utility functions have a greater impact on the adaptation decision as they directly affect the selection of variants. Furthermore, utility functions can be assumed to be highly *application dependent*. This is because, in practice, when these adaptation-behavior aspects of an application are defined, they are often custom-tailored to the applications themselves, which makes them hardly reusable. On the other hand, and similar to context reasoners, these artifacts are mostly associated to higher-level context types only, which makes them highly *platform independent*. Thus, it can be assumed that the context-aware applications, along with the self-adaptive behavior encoded in the utility functions, can be transparently ported to various deployment platforms.

4.3 Development methodology

As it is argued in [14], although *ad-hoc development* is still practiced by many organizations (especially in small-scale projects), it is generally considered as a chaotic and risky approach. The reason for this is that it relies heavily on the skills and experience of the individual staff members performing the work. In contrast to ad-hoc development, various methodologies have been proposed, such as the *waterfall*, *iterative* and *reuse* models (see subsection 2.1.2). It is reasonable to assume that for large and/or complex projects, using a disciplined and methodical development approach improves the quality of the resulting software while also reducing the required time and the cost associated with its development.

This development methodology is custom-tailored to facilitating the development of context-aware applications. As will be argued in the next section, one of the main components of this approach is its ability to *reuse* existing software components. Thus, the proposed methodology is based on the *reuse model* (see subsection 2.1.2), which is well suited for object-oriented and component-oriented computing environments [14].

An overview of the development methodology is illustrated in figure 11. In principle, this methodology is similar to the iterative development model (see subsection 2.1.2). However, in order to keep the description simple, steps related to the development of the business logic are omitted, and only steps relevant to enabling context-awareness are presented.

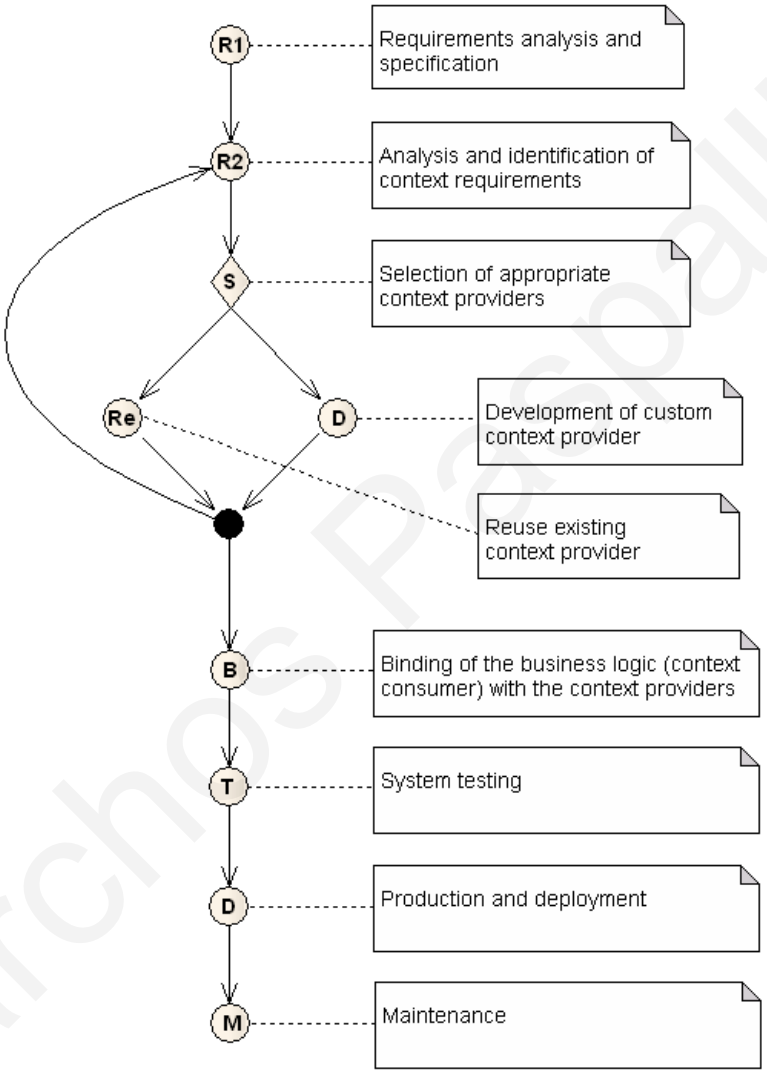

Figure 11: Overview of the development methodology

The first step (R1) refers to the requirements analysis of the application. At this point, the main components of the application are identified and listed. In the next step (R2) a more thorough analysis is performed concerning the *context requirements* of the application. The outcome of this

step is a list with the needed context types. In this case, context types are assumed to include both asynchronous context value notifications (i.e., time-triggered or value-triggered) as well as explicit context access on-demand.

Once the requirements step is completed, the developers proceed to realize the context-aware logic of their application. This phase involves the selection or implementation of context providers which are used to collect or generate the required context information. For example, if the context-aware application involves starting and stopping media playback according to whether the user is available in the room or not, then this can be simply realized by having a context provider notify the application when there is a change to the corresponding context type. A more complex example would be one where the location of a set of users matching a specific profile (e.g., located in the same building and having the same music interests) is retrieved and used in another application. Such an application would require context providers for sensing the location of the users as well as for providing access to their profile.

This step starts with a selection step (S) where the developer chooses if the corresponding context type will be provided by an existing component or by a newly developed one, which is custom-designed and implemented. This selection step mainly aims at facilitating software reuse. When the corresponding context type is a low-level one, like *location*, then it is expected that multiple context providers will be available, perhaps covering a wide range of deployment platforms. On the other hand, when the required context type is more sophisticated, like the *user-in-the-room* (indicating whether the user is in the same room as the device hosting the application), then it is more likely that a custom provider will be needed.

When the reuse step (Re) is selected, the developers select an existing provider (i.e., from a component repository), which is already implemented and packaged so it can be readily used with typically *no* glue code at all. Some context providers might be customizable in which case the

developers also have the chance to adjust the provider to the needs of their particular deployment. For example, the *polling interval* of a "GPS sensor"could be configured accordingly through a parameter specified in its metadata.

On the other hand, when the development step (D) is selected, then the corresponding context provider is designed from scratch either manually, as described in chapter 5, or with the help of appropriate development tools, such as the MDD-based approach described in chapter 6.

As some context providers can have additional context needs, it is required that the analysis and identification of context types (step R2) is revisited until all needed context types (including those required by the newly added—existing or defined—providers) are mapped to appropriate context providers. Once this is completed, the developers proceed to the binding of the business logic to the context providers (step B) as it was presented in subsection 4.2.3.

This is followed by the typical steps of system testing (T), which in this case must be extended to cover testing of the context-aware behavior of the application, besides its business logic. Once the tests are completed successfully, the application enters production and deployment (D), which is eventually followed by the maintenance phase (M).

At this point it should be noted that feedback from the testing phase might result into re-visiting some earlier phases (e.g., in order to pick a different, more suitable context provider). Furthermore, as the production and deployment phase is followed by the maintenance step, real-use feedback is expected to be collected and taken into consideration in order to adapt or evolve the application. Besides changes to the business logic, this might also result in requirements for updates to the context-aware behavior which would return the developers back to the requirements phase (R1 or R2). While these transitions are planned for, they are not shown in figure 11 to avoid cluttering.

4.3.1 Component repositories

One of the main advantages of Component-orientation is that it enables the use of software components in an *off-the-shelf* manner (COTS) [131, 65]. To facilitate this paradigm, many component repositories have been established. Such repositories are usually publicly available servers hosting a class of (usually relevant) software components. Access to the contents of such repositories can be achieved in any of three ways:

- *manually* (e.g., via web-access) where the developers browse the repository to select their preferred components and manually download them;
- *programmatically* by adding appropriate code in their applications to automatically search for and get the required components on-demand; and
- *interactively* where the end-users browse the repository via a console and use custom commands to search and get their selected components.

In the context of the proposed development methodology, a component repository is assumed to host *context providers*, thus facilitating their discovery and use. Such component repositories should allow the developers to search for available context providers based on some criteria. These criteria include primarily the requested *context type* and optionally additional extra-functional properties such as the *supported platforms*, the *fidelity of the provided context data*, etc.

As some context providers can have additional context dependencies themselves, those should be clearly identified in the repository, so that they can be taken into consideration during the selection process. In this regard, once a context provider is selected—as part of step (Re)—its required context types, if any, must be added to the list of required context types—step (R2) in the methodology. Examples of how this methodology—and the component repository—are used for the development of context-aware application are presented in chapter 7.

4.4 Context modeling

Context modeling refers to the approach used for abstracting and encoding the context information in an unambiguous form. An appropriate context model should facilitate data transfer, storing and network communication. Also, it should provide an unambiguous definition of the context artifacts, their representations, semantics and usage, while taking into account the general characteristics of context information, such as its temporal nature, ambiguity, impreciseness and incompleteness. Furthermore, a context model should also address special requirements of pervasive computing environments, such as distribution, mobility, heterogeneity of context sources, resource-constrained devices, *etc*. As it was discussed earlier, pervasive applications often require high-level context information that is derived from low-level context values. Therefore, an appropriate context model should also provide support for context reasoning. The rest of this section describes a context model which aims at satisfying these requirements.

4.4.1 Layered context model

This thesis assumes a rich context model which integrates sensed, static, user-supplied (i.e., profiled) and derived information—an assumption that is also used in the literature [68, 117], as it is considered useful and practical. Furthermore, in the process of making the context model as platform-independent and as reusable as possible, three layers are identified: *conceptual*, *exchange* and *functional*.

These layers are illustrated in figure 12 and are described as follows:

- The *conceptual* layer aims primarily at application developers. It includes an *ontology* and a *context meta-model*. The former guarantees both technical (i.e., structural) and semantic consistency across different platforms, while the latter is mainly used to facilitate

Model-Driven Development (MDD) approaches (see chapter 6). The conceptual layer also enables the definition of data-structure artifacts—such as elements, scopes, entities and representations—based on standard specification languages like the *Unified Modeling Language* (UML) and the *Web Ontology Language* (OWL) [13].

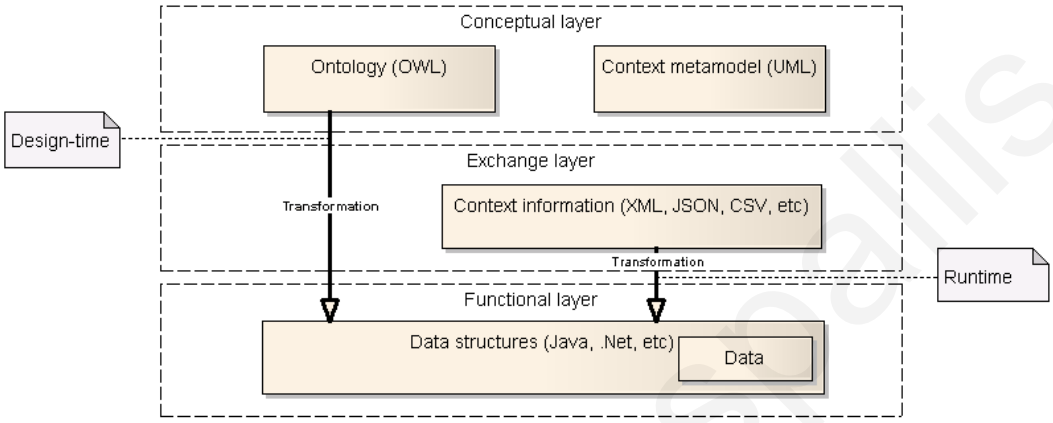

Figure 12: Layers of the context model

- The *exchange* layer aims to be utilized for interoperability between devices. In this regard, this layer is responsible for modeling the context data appropriately, so that they can be communicated across networked devices. To achieve this, appropriate mechanisms are made available for transforming context elements (see subsection 4.4.2) from their internal representation (as objects) into a serialized form, such as the *Extensible Markup Language* (XML), the *JavaScript Object Notation* (JSON) or Java's serialized stream form (see also subsection 4.4.4).

- Finally, the *functional* layer is responsible for the actual implementation of the context model, and the internal data-structures and mechanisms which are used to manipulate it. This layer can be object-based, but it does not necessarily need to be interoperable as it is platform-specific and, also, it is assumed that different devices might use different implementations of it (for example based on Java or .NET). As the main objective of this layer

is efficiency, the specific characteristics of each platform are taken into consideration when implemented. In most modern platforms (e.g., Java and .Net), the implementation is object-based.

4.4.2 Context meta-model

A context meta-model describes how general context information can be modeled. As such, it provides a view of the basic structures that can be used in defining specific context models. This thesis proposes a flexible context meta-model, the foundation of which is illustrated in figure 13. Context information is identified by two concepts, the *entity* and *scope*. These concepts are used to first disambiguate the entity to which the context information refers to (e.g., the *person*, *device*, or *environment* involved in the interaction) and second to identify the exact attribute of the selected entity that they characterize (e.g., *activity* of a *person*, *location* of a *device* and *WiFi signal strength* in an *environment*).

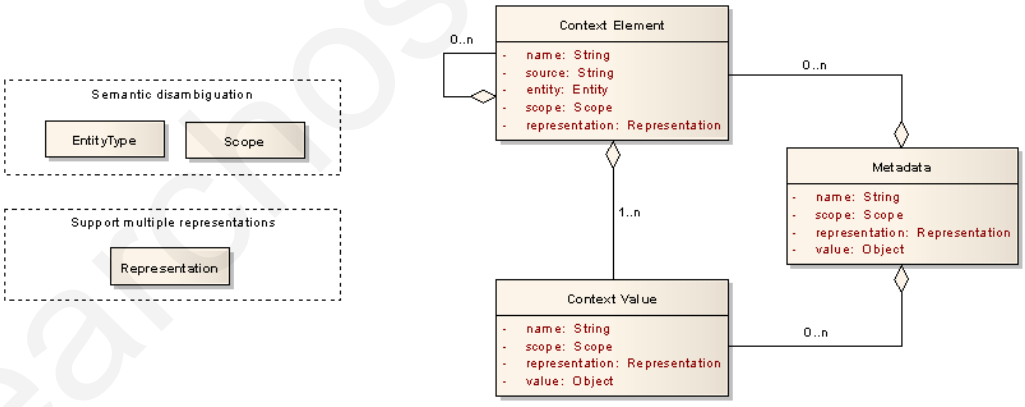

Figure 13: Context meta-model

Furthermore, the concept of *representation* is used to specify the internal representation used to encode context information in data-structures, effectively enabling alternative representations for the same context information. For example, the *temperature* might be represented either in

Celsius or Fahrenheit degrees. Also, the *location* of a *device* might be represented either in terms of coordinates (i.e., longitude and latitude) or in terms of a physical address (i.e., street, number, postal code, city, and country). Supporting transformation between multiple representations is an important requirement for enabling heterogeneity and interoperability. All these three concepts are highly related to the context ontology (see subsection 4.4.3).

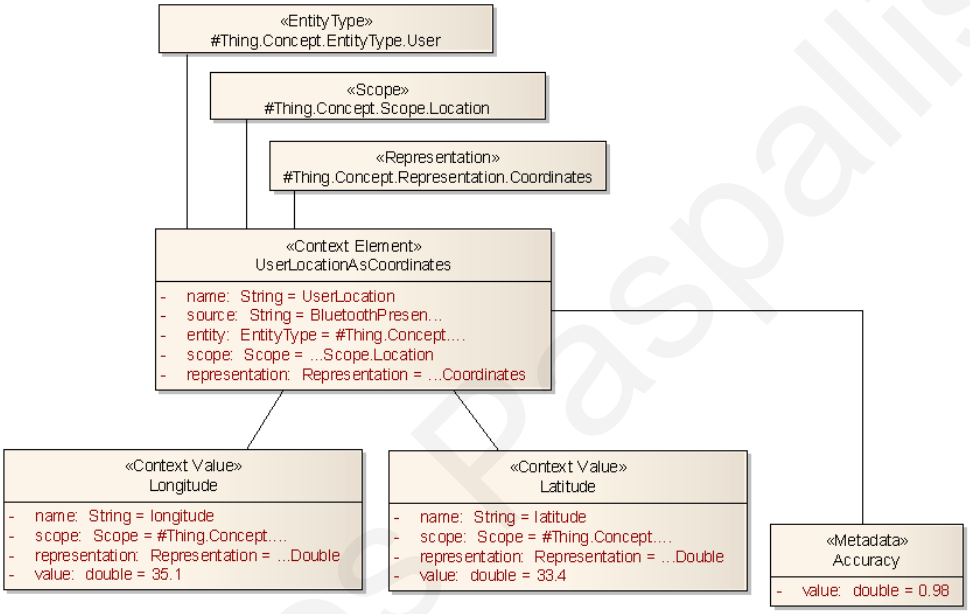

Figure 14: An example of the context model: showing "user location" expressed as coordinates

From a structural point-of-view, context information is organized in *context elements*, which group together information related to context entities. Context elements are composed of zero or more (child) context elements. Also, they must contain one or more *context values*. Self-nesting of context elements allows for expressing hierarchical relationships between context entities (e.g., a *device* entity might include additional entities characterizing its *memory*, *CPU* and *network* subcomponents). Context elements however, do not contain any direct context data. Rather, the context values are used as the basic means for encoding attributes of the context entities, which correspond to specific scopes. Finally, both context elements and context values can be associated

with none or more *metadata* values. Metadata are similar to context values, but they are used to characterize extra-functional aspects of context information, such as *accuracy*, *confidence*, *freshness*, etc. Instead of defining a predefined set of metadata, this open model was selected in order to enable developers specify arbitrary, custom metadata types which might be more relevant to their own applications.

For example, consider a context element which is used to encode information describing the location of a person. In this case, the entity is identified by the *entity*, *scope* and *representation* concepts, as per the context meta-model. The first two concepts state that the context information refers to a person (entity) and, in particular, to their location (scope). The representation also specifies that this model refers to a coordinates-based representation of location. The actual data of the location is encoded in two context values, corresponding to the location's longitude and latitude. Each of these values has a scope which corresponds to longitude and latitude respectively. The representation of both context values is of the `double` type, and in this example their value is `35.1` and `33.4` respectively. This context model instantiation is illustrated in figure 14 (the longitude and latitude values' scope and representation are not shown to avoid cluttering).

4.4.3 Context ontology

The context model is backed by an ontology. As shown in figure 12, the ontology is used at the conceptual layer during design-time in order to provide the developers with a consistent view of the available concepts (entities, scopes and representations). In other words, at this level it acts as a common dictionary among the developers, facilitating consistent use of context type identification during the development of context provider and context consumer components. Furthermore, the ontology facilitates dynamic transformation of context data from one representation to another at run-time. This is further discussed in the following paragraphs.

The basic concepts in the ontology are illustrated in figure 15 and are—naturally—consistent with the concepts of the context metamodel (see subsection 4.4.2). These are the `EntityType`, `Scope` and `Representation`. As the ontology is based on the *Ontology Web Language*, all three concepts inherit the properties of the `owl:Thing`, which is the standard root element in OWL. Central to this model is the `Scope`, which corresponds to any computable context value like temperature, social relationship, location etc. As such, each `Scope` can be associated to one or more `EntityTypes`, which it *characterizes*. Furthermore, each `Scope` is also associated to one or more `Representation` which is used to encode the *provided information*.

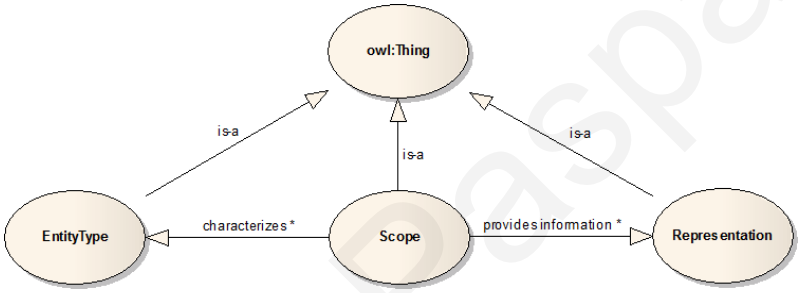

Figure 15: The basic concepts on the context ontology

The use of the ontology is introduced through an example. This example does not claim completeness, but rather aims at illustrating the general modeling concepts by describing how the conceptual layer, which contains the context meta-model, is complemented by the ontology. In order to provide an extensible ontology that is well-structured and easy to understand, a two-level hierarchy of the ontology, similar to the one presented in the *Service-Oriented Context Aware Middleware* (SOCAM) [136, 145], is introduced.

The context meta-model is linked to the ontology by the following three concepts: the scope that is characterized by the context element (*context scope*), the type of the particular entity of the characterized scope (*entity type*) and the *representation* of the context information.

Figure 16 presents the classes corresponding to the semantic concepts that need to be characterized through context information in the context management system. This example only illustrates a small number of classes, such as the concept `DateTime` which is a subclass of `BasicConcepts`. As depicted in this figure, the most important relation is that each `Concept` *has* a `Representation`. The class `Concept` is used to classify `EntityTypes`, `ContextScopes` and `BasicConcepts`. Additionally, some further relations between these classes and their subclasses can be defined (e.g., `isLocatedIn`). These relations can be used for ontology reasoning.

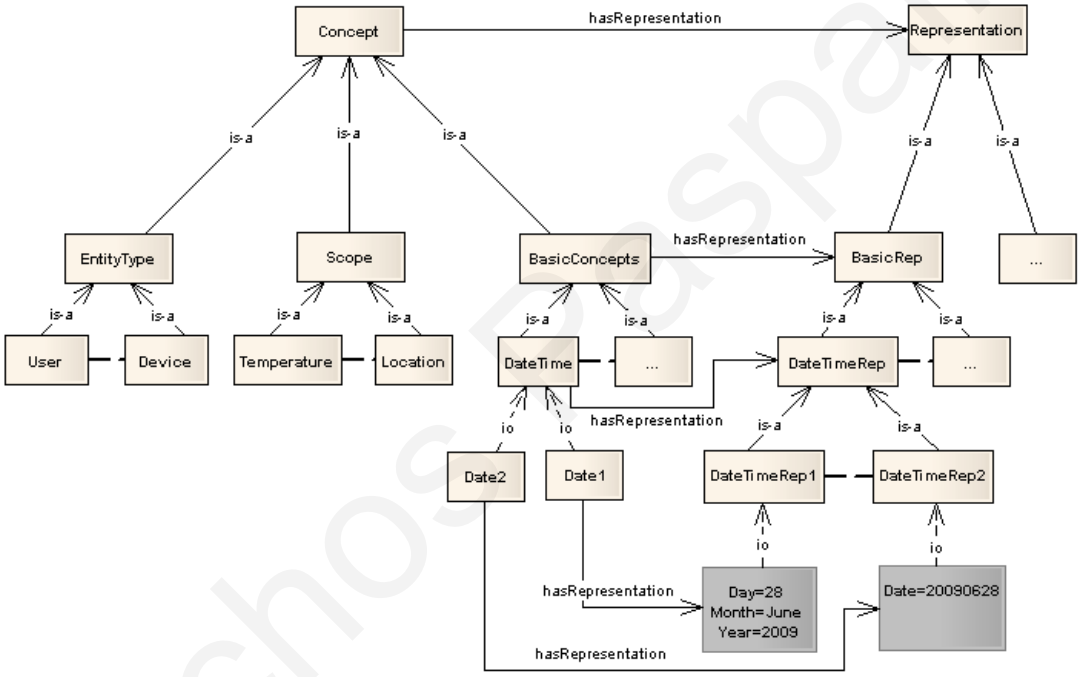

Figure 16: An example of the context ontology

As a second part of the ontology, the representations for the concepts must also be specified. As depicted in figure 16, a concept can have one or more representations. By allowing representing certain context information in multiple ways, not only is the challenge of heterogeneous context sensors for a certain semantic concept faced, but also the merging of ontologies is eased—at least to a certain extent. If an ontology matches a second one with regard to the classes for the concepts

and their relation, and they only differ in the representation of context information, the second ontology can be integrated in the first one in a straight-forward manner.

Finally, in order to enable explicit support for heterogeneous representations of context scopes, *Inter Representation Operations* (IRO) are enabled as they are defined in the ASC model [130]. This concept is a further step in supporting context providers and context consumers (see subsection 4.2.2) in a heterogeneous environment. For instance, it allows querying context information by describing the requested scope and entity, and by specifying a certain representation. If the available context information does not match the representation provided by the installed context providers, then the returned information is automatically transformed at run-time using the IRO functionality, as it is illustrated in figure 16.

This method is suitable for straight-forward transformation of context types such as in the case of date-time (e.g., between XML format, human-readable format and Unix time),[2] temperature (e.g., between Celsius, Fahrenheit and Kelvin), or speed (e.g., between kilometers per hour, miles per hour and meters per second). It should be noted that in all these examples the transformation can be easily realized in terms of simple functions which are often provided directly by the underlying ontology.

However, in some cases transforming the representation of a context type to another is not as trivial. For instance, it might be desirable to allow for *location* information which is alternatively represented in either of a coordinates-based or an address-based representation, as it is illustrated in figure 17. In this case, the transformation can be achieved either by utilizing a local database containing the appropriate mapping information, or via an online service which provides the equivalent functionality.

[2]Unix time, or POSIX time, is a system for describing points in time, defined as the number of seconds elapsed since midnight proleptic Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds.

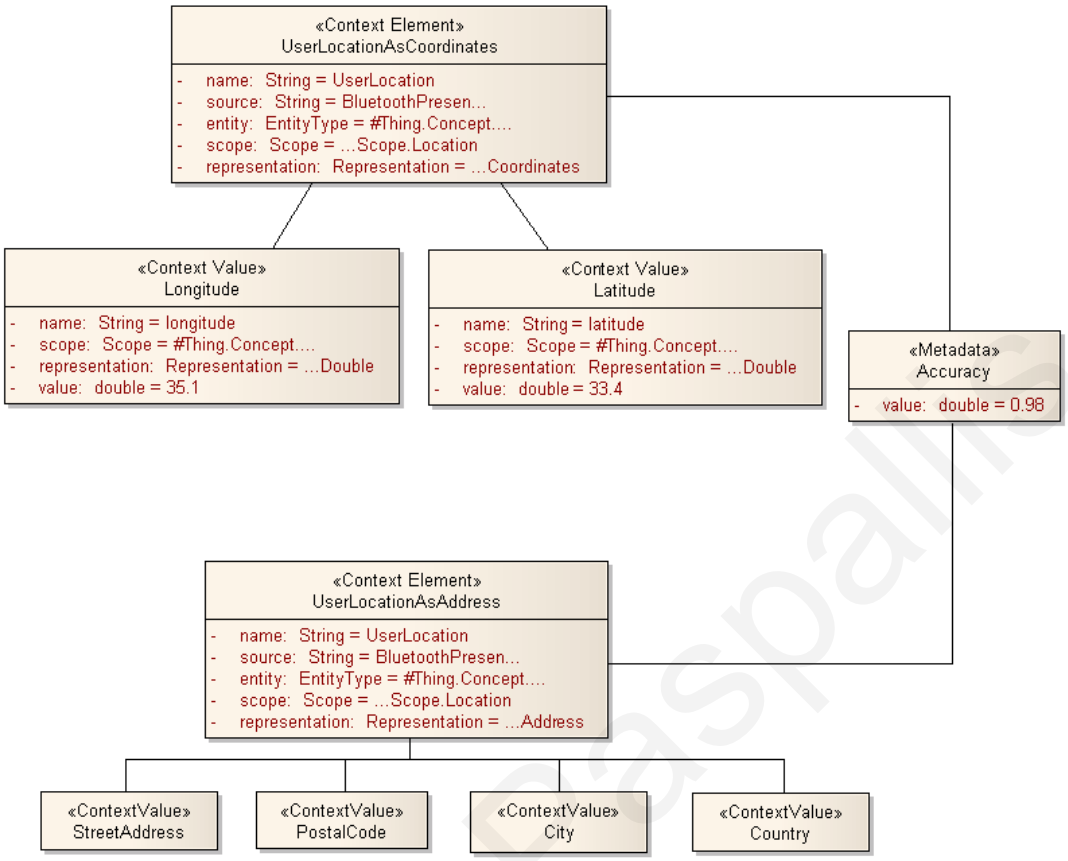

Figure 17: An example of the context model, illustrating two context elements corresponding to the same entity and scope but encoded with different representations

In this example, the two possible representations for modeling the location of a user are abstracted by two context elements respectively:

- The `UserLocationAsCoordinates` context element abstracts the location of the user as a combination of their longitude and latitude. These two values are themselves abstracted in two context value structures, namely the `Longitude` and `Latitude`.
- The `UserLocationAsAddress` context element abstracts the location of the user as a physical address. The constituent values are represented by the `StreetAddress`, the `PostalCode`, the `City` and the `Country` context values.

When the location of a particular user is queried, specifying either of the two representations, the underlying system either returns directly the retrieved information (assuming it is available in the requested representation) or it attempts to automatically perform an IRO transformation. In the case of the latter, there are two possibilities: Either the transformation is undertaken by existing code in the *functional layer* (see figure 12) as it was dynamically generated from the ontology, or it is delegated to a service (either local, as in the case of the database-backed approach, or remote), as that is also specified in the ontology. In practice, when a new transformation is defined in the ontology, it results to additional code which is dynamically generated and included in the functional layer of the context management system.

4.4.4 Modeling support for context distribution

To support context distribution, the context model must be able to maintain semantic consistency and functional interoperability across devices. Semantic consistency is achieved via appropriate notations in the context meta-model, while functional interoperability is enabled through a dedicated *Application Programming Interface* (API) which is defined in the exchange layer (see subsection 4.4.1).

4.4.4.1 Semantic consistency

Semantic consistency refers to the ability of distributed peers to have a *common view* of the meaning of the communicated context information. For example, when context information describing the memory availability of the device is communicated to a remote peer, it should be possible for the latter to have an unambiguous view of the semantics of its context data (e.g., it

corresponds to the *unused physical memory*) of an *identified device*, as well as of its representation (e.g., it is abstracted by an integer and it is measured in kilobytes). This example shows that semantic consistency is required at three levels:

- the context type, which is identified by the combination of the entity type and scope (e.g., the *unused physical memory of the device*);
- the identification of the entity into consideration (e.g., as it is identified by an international phone number in the case of a smart-phone); and
- a common understanding of the context type representation.

The first level of semantic consistency is implicitly achieved via the use of a common ontology. It is assumed that the developers build their systems using a common ontology document as a point of reference. This also allows for interoperability of components created by independent developers (for instance, individual context providers and context consumers—created by individual development teams—should be able to seamlessly interoperate when they are required to exchange context data).

The second level of semantic consistency is achieved by enriching the context model (at runtime) with extra information disambiguating the referred entities. For instance, an application receiving information about memory availability can assume that this value corresponds to the local device it is deployed on. However, in some cases, applications need access to context information characterizing distributed devices. For example, the case studies described in [18, 19, 105] require access to context information describing resource availability in connected—server—devices in order to offload some of their components to them. Evidently, in these cases it is important to provide the means to disambiguate exactly which entity the context information corresponds to.

Disambiguation is achieved with the use of so-called *groundings*. These allow to explicitly model *context entities* (e.g., users, devices, places) by using extra information identifying the corresponding entity (e.g., an email address for a user, an IP address for a device or coordinates for a place). In the example with the *unused physical memory* of a device, when the latter is identified as `#Thing.Concept.EntityType.Device`, then it is implicitly assumed to be the device running the middleware. But, when a grounding is specified, its identification is globally disambiguated as `#Thing.Concept.EntityType.Device|10.16.20.145`.[3]

Similarly, when a person is not identified, as in `#Thing.Concept.EntityType.User` (see figure 14), it is assumed that the model refers to the implied user (whose profile is loaded in the system and presumably uses the device). However, this is not sufficient when many users are relevant in the context of application. In this case, users are explicitly identified with a grounding, by providing a unique ID such as an email address. For example the author of this thesis is uniquely identified as `#Thing.Concept.EntityType.User|nearchos@cs.ucy.ac.cy`.

At this point, it should be noted that the context model does not limit the use of entities to just explicitly defined ones, as that would result in a much more strict query language that would prohibit the construction of general-purpose context-aware applications. For example, consider an application that requires access to the user's agenda. Apparently the user's entity, who is identified by a specific email address, can not be hard-coded in the context-aware logic. Rather, the model should provide support for late-binding of this sort of dynamic variables. For this purpose, the context model supports the use of two special-purpose grounding keywords: "myself" and "this". The former is used to characterize a user entity, which is dynamically resolved to the actual user ID at run-time, based on the user profile loaded in the system. The latter is used to characterize

[3]This notation is used to represent XML data. `Thing` is always the root element, and the `Concept` is used to indicate a specialized concept. The possible concepts (at this time) are the `EntityType`, the `Scope` and the `Representation`. The "`|`" symbol is used to indicate a value inside the corresponding XML tag.

the device where the middleware is deployed, and is also dynamically resolved at run-time using its domain name (if any) or its IP address.

Finally, the third level of semantic consistency is also enabled through the ontology, which provides means for expressing the actual representation of the modeled context information (see subsection 4.4.3). For example, the date-time metadata information can be alternative modeled in the form of a single string (e.g., "20090628"), or in a more granular composite form as three strings (e.g., "2009", "06" and "28"), as it was illustrated in figure 16.

4.4.4.2 Communication interoperability

Having established semantic consistency, context distribution also requires communication interoperability, enabling physical exchange of context information. As it was described in subsection 4.4.1, at the functional layer, context information is assumed to be modeled with actual *objects*, an approach that is natively supported in most modern platforms (including Java [22] and .NET [15]). In this regard, physical communication of context can be easily realized using mechanisms that are natively provided by the underlying platform (e.g., *Remote Method Invocation* (RMI) in the case of Java, and *Remoting* in the case of .NET).

However, native serialization is often inefficient as it is designed to encode arbitrary types of context, resulting to a processor-intensive operation. But, most importantly, this kind of physical communication is not platform independent, which makes it difficult for systems built on different platforms (e.g., Java versus .NET) to interoperate. To allow for a more efficient and platform independent approach, specialized code is provided, in the form of a dedicated API, allowing for conversion of context information modeled as run-time objects into a set of commonly-used formats including XML and JSON. This functionality is realized in the *exchange* layer, which was described earlier in subsection 4.4.1.

4.5 Context query and access

Having defined the model used to encode context information, the next step is to specify the means to access it. Direct context access is enabled through a simple API which allows context clients to query for specific types of context. The queries specify the entity and scope of the requested context type. Optionally, a representation is also specified, in which case the model either returns directly the resulting data—assuming it is modeled in the desired representation—or transforms it to the requested format using IRO whenever possible (see 4.4.3). Synchronous context access (on-demand) and asynchronous notification (value-triggered or time-triggered) is also made possible. In the first case, the called method returns directly with an object containing the requested context information. In the latter, a reference of the requesting object is passed along in the query, which is used to communicate context results—encapsulated in appropriate events—back to the client as needed.

4.5.1 Context query language

While the simple API approach is sufficient for simple cases where specific context types are needed, a more elaborate query language is needed when further filtering is required. By defining a rich query language, it is possible to encode much of the context-awareness logic of the client in a query, thus shifting the responsibility and complexity of realizing the context-aware behavior onto the query system. The latter parses the query, accesses the relevant context data, processes it and encodes the result accordingly.

This section presents a *Context Query Language* (CQL), which complements the context model (presented in the previous section) by enabling conditional and filtered context access. The CQL provides a rich language for specifying context queries, and thus granularly accessing

relevant context information while facilitating both synchronous and asynchronous modes. This section provides a brief overview of this language and its most important features.

The CQL is based on XML and is strongly related to the underlying context model and ontology. The main structure of a query is given in an enclosing XML element (i.e., `ctxQuery`). This element defines sub-elements for specifying the relevant Entity and Scope. Furthermore, it encloses the action element which specifies the type of the query, along with a set of conditions (i.e., `cond`) and constraints (i.e., `constraint`) which provide the required filtering.

The XSD schema of the XML-based context query language is shown in figure 18. The `contextQL` is the root of the XML-based document, which might be used to include one or more queries, defined as `ctxQuery`. Each query specifies exactly one `action`. There are two actions currently available in CQL:

- `SELECT` is used for specifying immediate access to the required context data, for a given entity and scope and for one or more specified conditions; and
- `SUBSCRIBE` is used to allow context clients to subscribe for asynchronous notification of context changes, for a given entity and scope, under some predefined conditions.

For actions of the `SUBSCRIBE` type, the subscription is limited to a predefined time interval, as stated in the query via the `validity` element—which is mandatory for this type of actions—and it can be renewed periodically. A unique identifier, marked as `subId`, is assigned to every subscription, and can be used by a context client to *renew* or *cancel* the subscription. Finally, a request for deleting a subscription can be placed through a `SUBSCRIBE` request, containing the subscription identifier and with the *validity* time set to zero.

Each query also specifies the `entity` and `scope` of the context type in question. Optionally, the query also includes a `validity` field to specify when is the query expired, and

a `subid` which is used as a unique identifier for as long as the subscription is active. Finally, the `timerange` element is a filter specifying the time period in which we are interested (i.e., return context values that occurred only within the specified period).

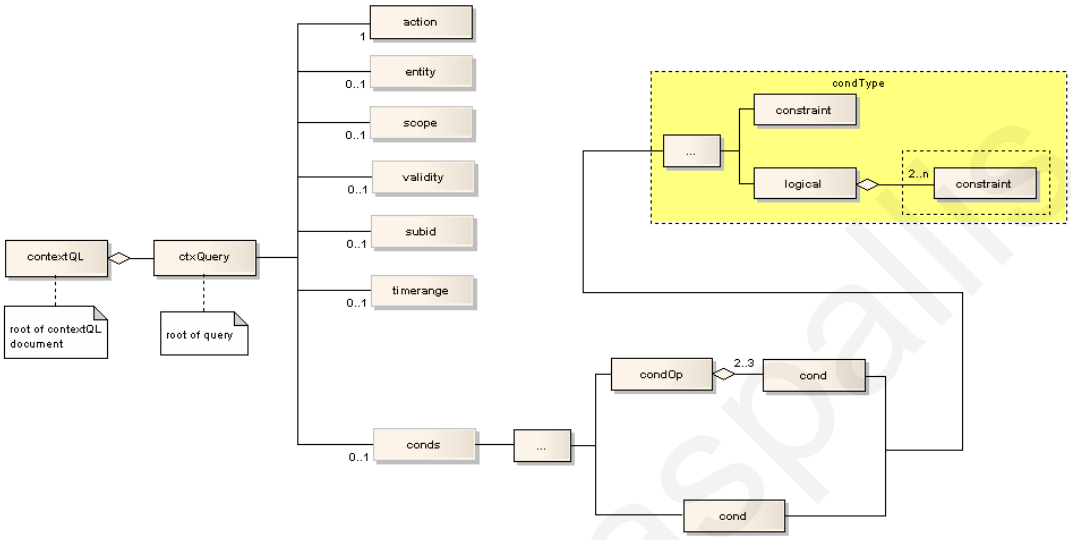

Figure 18: The XSD schema of the *Context Query Language*

The last element in a query is the `conds`, which is also an optional one. Nevertheless, this element is very important as it provides the means for specifying the *conditions* under which the context values should be retrieved. The `conds` element might include a single condition (defined as `cond`) or multiple conditions combined with some logical operator (defined as `condOp`). The latter combines the aggregated conditions (2 or 3 of them) using a logical operator such as `AND` and `OR`. Finally, each condition (defined as `cond`) is further analyzed into one or more constraints (see `condType` in figure 18). In case more than one constraint is defined, then those are also combined using a `logical` operator, such as `AND` and `OR`.

The `conds` elements defines additional filtering criteria for the selection of context values. In particular, three types of criteria are supported:

- `ONCLOCK` is used to specify that the context information should be obtained periodically, independently of any other constraints;
- `ONCHANGE` is used to specify that the context information should be obtained anytime there is an actual change in the context data; and
- `ONVALUE` is used to specify that the context information should be obtained only under the specified value conditions.

Furthermore, `constraints` are distinguished in *ordinary constraints* and *reasoning constraints*. An ordinary constraint defines a certain parameter and can thus have one or more of the following attributes:

- `par` is used to specify the name of the parameter which the constraint refers to;
- `value` is the value of the parameter;
- `delta` is used to specify a *threshold*—in terms of bounds—in cases where the constraint is a continuous number, thus enabling more flexible selection of data;
- `op` is the operator to be applied on the parameter for constraint evaluation (see table 2).

Table 2: Constraint operators of the CQL

`GT`	Greater than	`NCONT`	Not contains
`NGT`	Not greater than	`STW`	Starts with
`LT`	Lower than	`NSTW`	Not starts with
`NLT`	Not lower than	`ENW`	Ends with
`EQ`	Equals	`NENW`	Not ends with
`NEQ`	Not equals	`EX`	Exists
`CONT`	Contains	`NEX`	Not exists

A reasoning constraint is indicated through the tag `reasconstraint`. It defines a constraint on entities making use of the relationships between entities defined in the ontology, and has the following attributes:

- `relation` is used to define the relationship among the entities that are used in the constraint;

- `toEntities` is used to define the set of entities with which the returned entities have the specified relations.

The query language supports the resolution of ontology relations, such as *isChild*, *hasBrother*, *hasSister* and *isRelativeOf*. Furthermore, the query can specify a certain period of time using the `timerange` element. This tag has two attributes:

- `from` is used to identify the initial (i.e., beginning) timestamp of the context values requested in the query; and

- `to` is used to identify the final (i.e., ending) timestamp of the context values requested in the query.

If the `timerange` element is not present, it is implied that the query refers to the current (i.e., latest) context value. Finally, if the two attributes (i.e., `from` and `to`) are the same, then a single point in time (i.e., an instance) is implied.

An example of an XML-based query is illustrated in listing 4.1 (adapted from an example in [89]). As it is shown in this example, the query consists of a number of parameters which specify the context type via an entity and a scope. The context types are strongly coupled to the model through parameters which specify the XPATH (a special language used in XML to select nodes) of the corresponding concept in the ontology. For example, in the case of the scope of the requested type, the `ontConcept` refers to the corresponding semantic concept defined in the ontology, and the `ontRep` refers to the requested representation, which is also defined in the ontology.

The action element identifies which kind of operation is performed by the query through its type attribute (while `SELECT` is the most typical option, the `SUBSCRIBE` and `UNSUBSCRIBE`

options are also provided to indicate to the underlying context management system that the context client intends to subscribe or unsubscribe for asynchronous notification on changes to the specified context type). Finally, the action element is further customized with special conditions (i.e., `cond`), which allows filtering of the input context data (in this example, only addresses which have "Cyprus" in the country field are considered; thus the code in listing 4.1 queries the author's home address in Cyprus as he is identified by his email address).

```
<ctxQuery resultName="myHomeAddressInCyprus">
  <entity ontConcept="#Thing.Concept.Entity.User">
    nearchos@cs.ucy.ac.cy
  </entity>
  <scope
    ontConcept="#Thing.Concept.Scope.HomeAddress"
    ontRep="#Thing.Concept.Representation.DefaultAddressRep">
    homeAddress
  </scope>
  <action type="SELECT">
    <conds>
      <cond type="ONVALUE">
        <constraint par="homeAddress.country" op="EQ" value="Cyprus"/>
      </cond>
    </conds>
  </action>
</ctxQuery>
```

Listing 4.1: Example of a query formed using the CQL

An example of using the CQL in terms of realizing specialized context sensors—producing context types such as the *location* and the *WiFi signal strength* prediction—is included in the SSP case study presented in chapter 7.

4.6 Discussion

This chapter described a development approach for the creation of context-aware applications targeting mobile and pervasive computing environments. Unlike brute-force approaches which

merge the extra-functional aspects of such applications together with their business logic, a modular approach is proposed. This methodology separates the concerns of designing and implementing the context-aware, self-adaptive behavior of an application from that of defining and realizing its business logic. This chapter also presented a novel context modeling and querying approach, which was designed to address a number of requirements.

Evidently, numerous approaches exist for the development of context-aware applications, as it was discussed in chapter 3. This subsection provides a more detailed discussion of approaches which are more related to methodologies for developing context-aware systems, and also for modeling and querying context. These are compared with the methodology, context model and context querying language presented in this chapter.

4.6.1 Methodologies

The development methodology by Dey, Abowd and Salber [46] was one of the first approaches to be proposed for the development of context-aware applications. These authors described the development methodology in terms of the steps followed in the creation of the *Conference Assistant* application. Their methodology was mainly used to identify context components, as they were inferred from the application requirements. In this process, the designers identified relevant *context entities* and, then, for each entity they identified relevant *context attributes*. Additionally, the developers were expected to provide *requirements* for all pieces of context information, such as *measurement units*, *naming scheme*, *granularity/fidelity*, etc. Having defined those, the developers then used the Context Toolkit [44] to realize the application by following a number of sequential steps as follows: First, they defined a context widget for each context information (type), and also they defined an aggregator for each context entity. Next, they used interpreters for generating higher-level context information and for converting context data to the appropriate

form/fidelity. Following that, they interfaced these interpreters with applications and, finally, the context information was consumed inside the applications.

In [68, 70] Henricksen et al proposed an extensive development methodology. Their approach comprised a set of steps, also followed roughly in sequential order: The first step included the identification/specification of the core functionality, followed by the identification of choice points (i.e., where context could be exploited). The next steps consisted of the analysis and modeling of the required context types, as well as the analysis and modeling of the default—and, also, sample—user preferences. This was then followed by the application design and implementation using a programming toolkit [70], the configuration of the software infrastructure and, finally, by the testing and refinement of the preference sets.

While the approach presented in this thesis shares many similarities with these two approachers, it also differs from them, mainly in terms of the requirements identified in section 4.1. For instance, the separation of concerns requirement is clearly inherently supported by the development methodology. Furthermore, by placing emphasis on consistent, well-defined interfaces and appropriate component repositories, the proposed development methodology enables the realization of the context-awareness logic of an application by either reusing existing context providers, or producing reusable versions of them when needed. This architecture also facilitates the requirement for dynamically added—and removed—context providers and context consumers. This is enabled because the architecture follows a loosely-coupled approach where interaction between the components is dependent simply on context needs and context offers. Context distribution is facilitated in terms of semantic consistency and functional interoperability. This is complemented by the context access mechanism which enables synchronous and asynchronous access to context information with the use of a rich query language.

While the Context Toolkit specifies specialized operators for context sensing and processing (i.e., the widgets, aggregators, interpreters, etc), it does not facilitate the reuse of integrated context sensors or reasoners. The latter, as shown by the proposed methodology, can directly produce the requested context types while also seamlessly handling tasks such as sensing, aggregating and interpreting context data. On the other hand, the methodology by Henricksen et al aimed at a rather comprehensive solution for realizing the context sensing and the context consuming logic of the application. In contrast, the approach proposed in this thesis aims primarily at making the context information available to context consumers, leaving its actual use as a separate task. The advantage of the proposed approach is that it offers a more flexible method for exploiting the context information in any way deemed appropriate by the developers. Finally, it is argued that neither of these two approaches satisfies the two fundamental requirements specified at the beginning of this chapter: enabling separation of concerns and facilitating dynamically added/removed context providers and context consumers.

4.6.2 Context modeling

Along with the development methodology, this chapter also described a context model which is used for the realization of context-aware applications. An important advantage of the proposed context model is that it aims for a holistic solution, covering both design-time and run-time aspects. This is backed by the layered approach (see figure 12), which treats context modeling at different granularity based on the intended use.

While its default implementation makes extensive use of ontologies—which are described in the *Web Ontology Language* (OWL) [13]—the proposed context model is not exclusively dependent on it. As it will be discussed in the next chapter, the functionality of IRO and semantic consistency can be also realized programmatically (albeit not the design-time support). This flexibility

is required when the deployment platform is not resourceful enough to support the computations required by a full ontology implementation, in which case a programmatic approach is more appropriate. The rest of this section assumes an ontology-based model, discussing the advantage of the resulting model.

The main reason—and a significant advantage—for using ontologies is that they establish a mutually agreed understanding for the semantics of the different context concepts in a heterogeneous pervasive environment. For this purpose, the context ontology is defined and used to describe all relevant context concepts: the Scopes with their corresponding Entities and their various Representations.

Similar to the *Aspect-Scale-Context* (ASC) model by Strang et al. [130], the proposed approach also features inter-representation operations thus facilitating automatic conversion between measure units and data-structures, as well as more complex conversions between completely different representations (for example a *location* expressed in coordinates to/from a human-readable address). To allow this kind of transformations, the ontology provides groundings to certain methods in a library or to certain services which provide the corresponding functionality. Besides, the ontology is also used to describe relationships between entities, e.g. "a child has a father and a mother", or "a room belongs to a building". This allows the description of semantically complex queries in a compact manner, as a middleware-based ontology reasoner is used to automatically resolve the relationships.

4.6.3 Context query language

The CQL provides support not only for accessing single context elements, but also for retrieving sets of context elements with just a single query. Furthermore, it offers various ways to specify filters and conditions. By using logical operators, conditions can be combined to more complex

filters, theoretically with unlimited nesting. Besides, query filters and conditions are also applicable for context subscriptions. For this purpose, the `SUBSCRIBE` and `UNSUBSCRIBE` actions and the `ONCLOCK` and `ONCHANGE` condition types have been defined. The subscription mechanism also comes handy when requesting context information in terms of a continuous data stream. A client subscribes for the stream using a custom context query, and whenever information satisfying the query is available, the client is asynchronously notified.

CQL is also capable of dealing with the common characteristics of context information. With the help of the `timerange` tag, clients can query for current and also past context information (stored in context histories). Therefore, context's temporal nature is considered, and dynamic context information can be handled as well. The underlying context modeling approach is not limited to certain context scopes. Rather, it is highly extensible as all concepts defined in the ontology can be requested. Therefore, the new CQL is inherently able to handle concepts like location, proximity and spatial relationships. As CQL is not constrained to certain representations, defining custom representations in the ontology is also possible. Furthermore, support is provided to include arbitrary meta-data in the context. These meta-data can also be referenced in a query to construct elaborate filtering mechanisms.

One of the main advantages of CQL is its seamless integration with ontologies. With the use of semantic references, reasoning on context data is supported and heterogeneity is explicitly addressed. The CML system (see subsection 3.3.11) supports powerful reasoning on context information, but ontologies have been proposed only to be included to cover some special aspects, as e.g. privacy issues. Furthermore, heterogeneity is not addressed at all. The MoGATU system [112] allows the semantic description of context and also enables reasoning about it, building on the *Resource Description Framework* (RDF). However, unlike the proposed CQL, it does not support heterogeneity which facilitates the description of context entities with different representations as

well as the definition of associated IROs. The proposed CQL also facilitates the request for context information in a specific representation. The underlying query processing system automatically performs the necessary conversions between different representations. A further shortcoming of existing approaches is the lack of appropriate sets of aggregation functions. Not only does CQL provide several predefined aggregation functions like `average`, `sum`, `max` or `min`, but it also provides support to define more elaborate aggregation functions along with their groundings in the ontology.

Finally, as the proposed CQL does not refer to specific instances of context sources, it is well suited for dynamic pervasive computing environments. Context sources may be locally deployed or remotely accessible, and they can appear or disappear dynamically. These characteristics—autonomy, mobility and distribution—are also explicitly supported by the corresponding context management system architecture (see chapter 5).

4.7 Conclusions and future work

This chapter presented a methodology for developing context-aware applications. The methodology eases the development task by separating it into the concerns of providing context information and that of consuming it. Also, through this process, the developers are enabled to reuse existing context plug-ins whenever possible, or build new ones when required. Furthermore, a layered context model was presented, providing support for semantic consistency and interoperability. The model was accompanied by a context query language which facilitates filtering of context data, thus shifting the complexity of context-aware logic into the middleware. The development methodology, the context model and the query language are evaluated in chapter 7 where their use is also illustrated through two case-study applications.

Although the development methodology and the context model have improved substantially since they were originally presented in [107] and [114], there is still room for improvement. For instance, an important addition to the development methodology is the realization of an approach which allows the development of complete context-aware, self-adaptive applications with reusable components. A first attempt at specifying a blueprint for such an approach was presented in [105]. This approach discusses how applications can be developed by separating the concerns of where, when and how adaptations apply. Nevertheless, in this approach only the functional aspects of the components are reusable (i.e., their adaptive behavior is strictly coupled to the original application).

A second important improvement concerns the context query language. In its current specification, CQL does not support *inner join* operations (i.e., where the results from a first query are further filtered by additional conditions referring to a second element). This inadequacy is show-cased in the case study application presented in section 7.2. In this respect, one of the future research directions involves the enhancement of the CQL specification to facilitate such elaborate and powerful queries.

Chapter 5

A pluggable and modular middleware architecture for context-aware applications

This chapter presents a middleware architecture which enables individual context providers and context consumers to be independently developed and dynamically composed via a middleware layer to form context-aware applications. The proposed architecture allows dynamically adding or removing context providers (such as a Bluetooth-based GPS sensor) at run-time, without requiring that the dependent applications (such as an interactive walking tour application) are restarted in order to take advantage of the additional—or richer—context information. Furthermore, by separating the roles of context providers and context consumers, and also by building on top of a component-oriented architecture [131], code reuse in the form of context plug-ins is facilitated. Also, by monitoring their metadata—which encode their *provided* and *required* context types—the middleware is enabled to intelligently and autonomously activate and deactivate context plug-ins as needed, thus optimizing their resource consumption (e.g., battery drain, memory use, etc). Finally, the architecture of the middleware itself is modular, allowing for alternative variants of it to be easily formulated, better matching the requirements and characteristics of the target domain.

5.1 Introduction

As it was discussed earlier, the proposed solution aims, primarily, at mobile and ubiquitous computing environments. While the importance of these environments is continuously increasing, they also introduce a number of constraints which in turn become challenges when faced by software developers. In this regard, a comprehensive solution is aimed to benefit both the developers at design-time and the users at run-time. A number of requirements were identified in section 3.2, partially to guide the development process. These requirements also serve for evaluating the conformance of the resulting middleware architecture to the predefined goals (see chapter 7).

The single most important, functional requirement of the middleware architecture is quite straightforward: provide application-specific access to context information. To this end, the architecture should provide support for *collecting*, *storing*, *organizing* and *accessing* context information, as it was discussed in section 4.1. Additional requirements, such as interoperability, inference of inter-dependencies of context and support for context distribution, are also important. These functional requirements have led to the implementation of a comprehensive context model and of an elaborate *Context Query Language* (CQL), which were described in chapter 4.

Besides these direct requirements, the nature of mobile and pervasive computing imposes some additional, mainly extra-functional ones. The study of these requirements provides valuable insight into the decisions which were taken during the design of the architecture.

5.1.1 Middleware architecture-specific requirements

This subsection revisits some of the requirements identified earlier in chapter 3, and examines them in more detail from the perspective of designing the middleware architecture.

- *Dynamic behavior*: A pluggable architecture enabling dynamic behavior, where plug-ins can be installed/uninstalled and activated/deactivated at run-time, is needed for several reasons. First, it facilitates separation of concerns as different components are used for sensing the context (i.e., producing) and others for using it (i.e., consuming). In this way, instead of hard-coding the context-aware behavior in different applications, the pluggable architecture allows defining it in individual context plug-ins which can be reused at design-time and shared at run-time. Second, such an architecture can facilitate multiple context providers being interfaced to multiple context consumers. This feature enables scenarios where multiple context sources are multiplexed to multiple applications at the same time. Finally, this kind of architecture allows for dynamic activation and deactivation of context plug-ins as needed, thus supporting more efficient resource consumption. This is particularly important for mobile and embedded devices which are generally characterized by limited resources and capabilities.

- *Modularity*: As different applications and different deployment platforms have different requirements and capabilities, a modular architecture is needed to ensure that those and only those features that are explicitly needed, and compatible with the target platform, are used. A modular architecture can enable this by allowing different variants of the middleware to be composed at development-time based on the capabilities and constraints of the target platform and applications. As the resulting architecture can be custom-tailored to these characteristics, its resource footprint on the deployed devices is minimized.

- *Platform independence*: Another characteristic of mobile and ubiquitous computing is the variety of devices and platforms they are deployed on. As such, it is important that the proposed solution addresses this problem by either employing completely platform-independent

components and protocols, or at least allowing for straight-forward, if not fully automated, porting of the relevant components. Assuming a modular architecture, then it is desired that the majority of the (core) components are platform-independent. Also, for those components that need to be platform-specific, it is desired that they feature alternative implementations.

- *Lightweightness*: This requirement stems from the need for both easy development of context plug-ins and limited resource availability in the target devices. However, although lightweight, the proposed architecture should be complete in terms of functionality. Also in relation to the modularity requirement, and taking into consideration the tradeoff between feature completeness and lightweight-ness, it is desired to have the right variants for the right deployment requirements.

- *Adoption of existing patterns and standards*: To facilitate both a smoother learning curve for developers and also to maximize the reuse potential of the existing codebase, it is important that the provided solution and its implementation follow and use existing patterns and standards whenever possible. While it is unlikely that any developer is familiar with all possible software architectures and methods, experienced programmers are often quite competent with frequently used patterns, such as those described by the *Gang-of-Four* in [52]. Similarly, while custom software systems—such as component frameworks—require significant effort to get accustomed too, adoption of widely used standards, such as the *Enterprise Java Bean* (EJB) containers [95] and the OSGi R4 framework, allow for nearly instant productivity.

5.2 Pluggable middleware architecture

In order to accommodate the requirements detected above—while also keeping in mind the requirements summarized in table 1—a single, basic guideline was adopted: The developers must be able to "*independently develop and deploy context providers and context consumers*". In other words, the concern of developing context-aware applications is separated into the concern of developing *context producing* components and the concern of developing *context consuming* components. This simple rule has led to the adoption of a *pluggable* architecture, where the context providers are designed and developed as *plug-in components* which are capable of independent deployment and activation. These components are referred to as *context plug-ins*. On the other side, the context-aware applications act as context clients, and are only loosely coupled with the context providers.

To support such separation of concerns, a middleware-based system is defined. This system acts as an intelligent context hub: it collects, stores, processes context data, and makes it available to context clients. It should be noted that in this architecture the context providers and the context consumers are *loosely coupled*, interacting with each other only through the middleware based on their context dependencies, as it was illustrated in subsection 4.2.3. This approach enables developers of context-aware applications to build their own context plug-ins or reuse existing ones, as per the proposed development methodology.

While some context plug-ins can be pure context providers (i.e., context sensor plug-ins), others may also require input of—typically—primitive context data and thus can be considered as both context providers and context consumers (i.e., context reasoners plug-ins).

The context types provided and—optionally—required by each context plug-in are explicitly defined at design-time and accessed by the middleware during installation. The middleware uses

this information to resolve the context dependencies of the plug-ins. Furthermore, based on the context types that the deployed context-aware applications need, the middleware dynamically activates an appropriate subset of context plug-ins. The corresponding mechanisms are described in detail in subsections 5.2.3 and 5.2.4.

This chapter presents a pluggable middleware architecture which realizes the conceptual model of context providers and context consumers presented in section 4.2. This implementation is based on the OSGi component framework.

5.2.1 Context plug-in lifecycle

To accommodate the lifecycle required by this approach, the OSGi component specification [132] is leveraged. OSGi provides support for designing and deploying software components (packaged in so-called bundles) as well as for connecting them to each other. The latter is enabled by allowing the deployed components to register their provided services and/or discover and consume services registered by other components.

OSGi supports a simple—yet powerful—component lifecycle, which allows for dynamic installation, update, resolution and activation of components. This is illustrated in figure 19, which depicts both the original OSGi component lifecycle, along with the extensions added to accommodate the mechanisms of the presented context middleware.

In its simplest form, the OSGi component lifecycle dictates multiple states and transitions. Initially, as soon as a component is installed, its state is set to `INSTALLED`. Resolving the static (i.e., package imports) and dynamic (i.e., service-based) dependencies of a component yields the `RESOLVED` state. At that point, the component is ready to be activated, in which case its state is changed first to `STARTING`, and then to `ACTIVE` (via an automatic state transition denoted with a dashed arrow). Stopping a component sets its state back to `RESOLVED`, after switching it

to the STOPPING state. While in either of the INSTALLED or RESOLVED state, a component can be uninstalled, which results in the terminal state UNINSTALLED. Furthermore, while in the INSTALLED or RESOLVED state, a component can be updated (so that its actual implementation code can be replaced). In both cases, this yields the INSTALLED state, in which the component must first be resolved again before it is readied for activation.

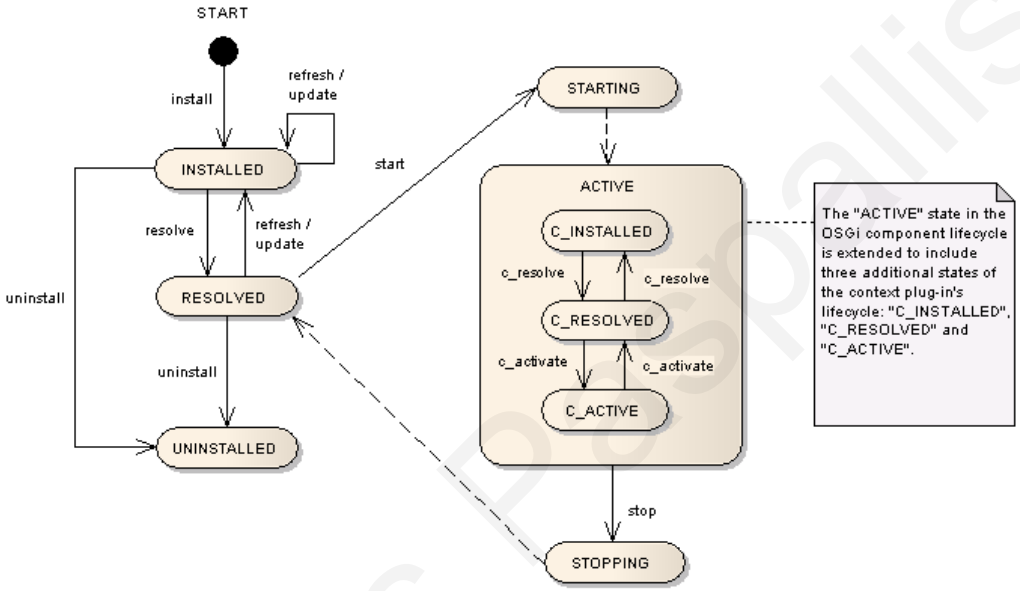

Figure 19: Extended OSGi component lifecycle state diagram

While the OSGi lifecycle is sufficient for resolving the static dependencies of bundles (i.e., static libraries used) as well as the dynamic dependencies between them (i.e., services used), it is inadequate for the purposes of the context middleware. The reason is that the deployed context plug-ins have additional dependencies which cannot be explicitly expressed as either library or service dependencies. These dependencies change dynamically as new plug-ins are installed or uninstalled and as new applications are started or stopped.

To overcome this limitation, the context plug-in lifecycle is amended with three context middleware specific states: C_INSTALLED, C_RESOLVED and C_ACTIVE. These states are not controlled by the OSGi framework, but rather by the context middleware. While in the ACTIVE

state, it can be safely assumed that the context plug-in has resolved both its static and dynamic dependencies. Specialized algorithms (see subsection 5.2.3) are used to monitor and control the plug-ins' extra-functional dependency on context.

As it was discussed earlier, the context plug-ins define their provided context types in addition to their required context types. In this respect, while context sensors are considered to be always resolved, context reasoners are considered as resolved *only when their context dependencies are satisfied*. For instance, a context reasoner plug-in which produces context type A and requires context type B cannot be resolved unless there is at least one plug-in installed—and *resolved*—offering context type B (transition from `C_INSTALLED` to `C_RESOLVED`). Similarly, a plug-in offering a context type C is activated only when there is at least one active context consumer in need of that context type (transition from `C_RESOLVED` to `C_ACTIVE`). As changes to the needed or provided context are dynamic, transitions back from `C_ACTIVE` to `C_RESOLVED` and possibly to `C_INSTALLED` are also possible. Finally, when a plug-in is stopped (e.g., so that it can be un-installed), it is first transitioned back to the `C_INSTALLED` state before it continues with the normal flow of the transitions dictated by the OSGi component lifecycle.

When a plug-in's state transitions to `C_ACTIVE`, then a special `activate` method is automatically invoked by the middleware. Similarly, when its state transitions out of the `C_ACTIVE` state, the `deactivate` method is invoked. This approach is in line with the *Inversion-of-Control* (IoC) pattern. The complete specification of a context plug-in is illustrated in listing 5.1 as an interface encoded in Java. The lifecycle-handling methods are the `activate` and `deactivate`.

The rationale for these context plug-in-specific states is the following: When a context plug-in is installed and resolved (in OSGi terms), it should only be activated when that would be useful. But, if the plug-in is not resolved (in context-dependency terms) then activating would not yield any context data as the required input would never be provided. Similarly, when no context client

requires the context types provided by a specific plug-in, then activating it will not have any impact to the system, except for resource consumption. Thus, by introducing these specialized states, it is possible to optimize the resources consumed by the context plug-ins by activating those, and only those, which are absolutely necessary.

```
public interface IContextPlugin
{
    public String getID();
    public PluginType getType();
    public IPluginMetadata getMetadata();
    public void setContextListener(IContextListener);
    public void setContextAccessService(IContextAccess);
    public void activate();
    public void deactivate();
}
```

Listing 5.1: Specification of a context plug-in

In practice, it is assumed that the developers implement their plug-ins so that when they are activated, they allocate their required resources (i.e., by overriding the activate method shown in listing 5.1), and when they are deactivated they de-allocate those resources (i.e., by overriding the deactivate method also shown in listing 5.1). For example, if the context plug-in is used to generate *location* context types via input from a GPS device, then activating the plug-in would result in turning the GPS sensor on and deactivating it would result in turning it back off. Consequently, by only activating the location plug-in on when an actual context-aware application needs this context information, it is possible to conserve significant resources (in this case battery consumption).

Finally, in addition to the core OSGi, the context middleware also uses the *Declarative Services* specification, which is itself based on the foundations of the *Service Binder* [35, 36]. This specification allows for explicitly defining the functional requirements and offerings of a component, in terms of services. In this approach, services refer to predefined interfaces, enabling

contract-based substitutability. Additional support is provided for defining both *optionality* and *multiplicity* for service bindings. Based on these metadata, the service dependencies are automatically checked and resolved by the OSGi framework at run-time, as needed. The context middleware architecture extends this specification by defining custom properties which describe the context types provided and required by the corresponding context plug-in. These metadata are specified in appropriate configuration files, such as those illustrated in listings 5.2 and 5.3, and are used by the middleware when resolving and activating the plug-ins, as it will be discussed in the following section.

5.2.2 Context plug-ins

While the core middleware architecture defines the framework for deploying plug-ins and serving context clients, the actual context sensing and reasoning logic lies within the context plug-ins. The plug-ins are defined as pluggable OSGi bundles, and they are realized as classes implementing the `IContextPlugin` interface (see listing 5.1). This interface provides the functionality to access the plug-in's metadata, which include the *required* and the *provided* context types.

These metadata can be either explicitly defined via a class realizing the `IPluginMetadata` interface or, they can be implicitly realized using an automated mechanism which reads the metadata from the plug-in's deployment descriptor. For example, the *Motion Detector*'s deployment descriptor, illustrated in listing 5.2, implies that the plug-in has no required context types and that it provides a context type described by the specified entity, scope and representation values.

After a plug-in is installed (and assuming it is resolved and activated in the OSGi terms), it is registered by the context middleware which records its *provided* and *required* context types. Additionally, when context clients connect to the context middleware and inquire context data, the latter also records data about these clients and their required context types.

At run-time, the required and provided context types are processed by the *context manager*, which uses this information to resolve and activate the plug-ins as needed. In particular, the resolution mechanism is triggered when the provided context changes, i.e., every time a new plug-in is installed or an existing one is uninstalled. The activation mechanism, on the other hand, is triggered when the required context changes, i.e., when a new context client is activated or an existing one deactivated. The result of this process is to activate those, and only those, context plug-ins that are resolved and offer context types needed by active context clients.

```
<component name="MotionDetector" immediate="true">
  <implementation class="cy.ac.ucy.cs.mtn_det.MotionDetectorPlugin"/>
  <service>
     <provide interface="org.istmusic.mw.context.plugins.IContextPlugin"/>
  </service>
  <property name="Provided-Entity" value="#Thing.Concept.EntityType.
      environment|room"/>
  <property name="Provided-Scope" value="#Thing.Concept.Scope.motion"/>
  <property name="Provided-Representation" value="#Thing.Concept.
      Representation.motion"/>
</component>
```

Listing 5.2: An example of a plug-in service descriptor (see *Motion Sensor* in subsection 7.1.5)

A two-phase approach is followed: the plug-ins are first resolved and then activated as needed. The corresponding algorithms used in these mechanisms are presented in the following two subsections, and their effectiveness is evaluated in subsection 5.7.2.

5.2.3 Resolution mechanism

The resolution mechanism is responsible to mark plug-ins as resolved or unresolved. Resolved are those plug-ins which either have no context dependencies, or their dependencies are offered by some other, resolved plug-ins. It should be noted that when a plug-in is marked as resolved, it is implied that it can be instantaneously activated and start producing events of its provided context

type. As it is evident from the definition, an iterative algorithm is a straightforward approach for realizing the resolution mechanism. This mechanism is depicted in algorithm 5.1.

Algorithm 5.1 The algorithm used by the resolution mechanism in pseudo-code

Basic data-structures
[all plug-ins] - set containing all installed context plug-ins
[resolved] - subset of the [all plug-ins]; contains only resolved plug-ins
[provided] - map of context types C_T to set of resolved plug-ins providing C_T
Triggered by
Changes to the [all plug-ins] set
Algorithm

```
1:  # first ensure that all resolved plug-ins are included in [resolved]
2:  changes-detected ⇐ true
3:  while changes-detected do
4:     changes-detected ⇐ false
5:     for all p in [all plug-ins] - [resolved] do
6:        if requiredContextTypes(p) ⊆ [provided].keys then
7:           [resolved] ⇐ [resolved] ∪ {p}
8:           [provided] ⇐ [provided] ∪ {providedContextTypes(p) → p}
9:           changes-detected ⇐ true
10:       end if
11:    end for
12: end while
13: # next make sure that no unresolved plug-ins are in the [resolved] set
14: changes-detected ⇐ true
15: while changes-detected do
16:    changes-detected ⇐ false
17:    for all p in [resolved] do
18:       if not requiredContextTypes(p) ⊆ [provided].keys then
19:          [resolved] ⇐ [resolved] - {p}
20:          [provided] ⇐ [provided] - {providedContextTypes(p) → p}
21:          changes-detected ⇐ true
22:       end if
23:    end for
24: end while
```

This resolution mechanism defines three data-structures: First, the `all plug-ins` is a set which includes all the installed plug-ins. As it was mentioned earlier, the execution of this algorithm is triggered by changes to this set. The second data structure—`resolved`—is a subset of the `all plug-ins` set, and contains the plug-ins which are marked as resolved. Finally, the last data structure—`provided`—is a map which associates each context type to a list of context

plug-ins. Each of these lists includes those, and only those, plug-ins that are known to provide the corresponding context type and which are known to be resolved.[1]

Regarding the algorithm, it consists of two phases: In the first phase, it makes sure that each resolved context plug-in is marked as such (i.e., it is included in the [resolved] set). To achieve this, it repeatedly checks each unresolved plug-in to see if there is a change in its dependencies. If it is found to be resolved it is marked as such. Otherwise, when a complete iteration of the unresolved plug-ins is completed without a change, it is assumed that all resolved plug-ins have already been marked as such. This part of the algorithm always terminates, because there is only a limited number of unresolved plug-ins, and at least one of them is shifted to the `resolved` set in each loop—with the exception of the last one.

The second phase achieves a symmetric goal: it ensures that all plug-ins marked as resolved, are indeed such. To achieve this goal, an iterative process is used again, where each plug-in that is marked as resolved is checked against its dependencies. If it is found to be unresolved, it is unmarked (deleted from the "`resolved`" set) and the process repeats until a full iteration is completed without changes. This part of the algorithm also always terminates, because there is only a limited number of plug-ins in the "*resolved*" set, and at least one of them is deleted from that set in each loop—with the exception of the last one.

5.2.4 Activation mechanism

While the resolution mechanism is triggered by changes to the availability of context plug-ins, the activation mechanism is triggered by changes to the context needs. As context-aware applications are started and stopped, they register and unregister for context notification with the context management system. For this purpose, the *context manager* maintains a mapping for each needed

[1] From a technical point of view, a context type consists of an entity/scope pair, as they were defined in section 4.4.2

context type, pointing to a list of registered clients. Thus, even context clients which require periodic, synchronous access to context data are required to register their context needs with the middleware (using a specialized method in the context access service). Asynchronous subscribers to context data are registered implicitly while placing their queries. This approach allows the context management system to be aware of the actual context needs of all active applications, and thus be able to intelligently activate and deactivate the context plug-ins as needed. The details of the activation algorithm are presented in algorithm 5.2.

This algorithm leverages four data structures: First, `provided` is a map of all provided context types to a list of resolved plug-ins providing them (i.e., this data-structure is identical to the corresponding data-structure used in the resolution mechanism). The second data-structure—`needed`—is a similar one mapping, however, needed context types to the corresponding requestors. In this case, the keys are context types and the values are lists of objects, corresponding to the context clients. Next, the `resolved` is a set containing all plug-ins that are marked as resolved and, finally, the `active` is a set containing all plug-ins that are marked as being active. Naturally, the `active` is a subset of the `resolved` set, as that was defined in the resolution mechanism. Updating these data structures is the main goal of this mechanism.

After an initialization phase where all unresolved plug-ins are removed from the set of active plug-ins, the activation algorithm spans two phases, just like the resolution one. In the first phase, all the plug-ins to be activated are selected by iterating the needed context types and *selecting* from the ones providing the corresponding context type. After the appropriate plug-ins are selected for the given context type, the "`needed`" data-structure is also updated with any new context types possibly needed by the newly activated plug-ins. This loop is repeated until no changes are detected.

Algorithm 5.2 The algorithm used by the activation mechanism in pseudo-code

Basic data-structures

[provided] - map of context types C_T to set of resolved plug-ins providing C_T
[needed] - map of all needed context types to lists of corresponding clients
[resolved] - contains those, and only those, plug-ins that are resolved
[active] - the set of all plug-ins that need to be active

Triggered by

Changes to the [needed].keys context types set

Algorithm

```
1:  # first ensure that unresolved plug-ins are removed from the [active] set
2:  [active] ⇐ [active] ∩ [resolved]
3:  # next make sure that the needed plug-ins are in the [active] plug-ins set
4:  changes-detected ⇐ true
5:  while changes-detected do
6:     changes-detected ⇐ false
7:     for all C_T in [needed].keys do
8:        for all p in select(provided [C_T]) do
9:           [active] ⇐ [active] ∪ {p}
10:          [needed] ⇐ [needed] ∪ {requiredContextTypes(p) → p}
11:          changes-detected ⇐ true
12:       end for
13:    end for
14: end while
15: # last make sure that no unneeded plug-ins are in the [active] plug-ins set
16: changes-detected ⇐ true
17: while changes-detected do
18:    changes-detected ⇐ false
19:    for all p in [active] do
20:       if providedContextTypes(p) ∩ [needed].keys == ∅ then
21:          [active] ⇐ [active] - {p}
22:          [needed] ⇐ [needed] - {requiredContextTypes(p) → p}
23:          changes-detected ⇐ true
24:       end if
25:    end for
26: end while
```

It should be noted that the *select* operation defined in line 8 of algorithm 5.2 can be as basic as selecting *all* the plug-ins providing the corresponding context type C_T, or, it can be quite complex featuring intelligent selection of a *subset* of the plug-ins based on their resource consumption and their *Quality of Service* (QoS) properties. This thesis presents only a basic approach, which was implemented, tested and evaluated as documented by Paspallis et al in [110]. However, providing more elaborate implementations for the selection operator is a promising future direction that is currently being investigated.

In the second phase of the algorithm, each plug-in that is marked as active is checked against the needed context types. If none of its provided context types is needed by the active context clients, then it is unmarked and any context requirements it possibly has are removed from the `needed` map. This process is repeated until no new changes are detected.

At the end of this algorithm, the states of the plug-ins are checked, and their lifecycle is adjusted accordingly. For instance, any active plug-ins that were unmarked are deactivated and any inactive ones that were marked are activated. From a technical point-of-view, the changes in the plug-ins' lifecycle are enabled via the corresponding methods defined in the `IContextPlugin` interface, presented in listing 5.1.

5.3 Modular middleware architecture

The context middleware is a component-based framework itself, consisting of a central basic sub-component—the *Context Manager*—and multiple secondary ones, including the *Context Distribution* and *Context Privacy*, the *Context Model*, the *Context Repository*, and the *Context Query Processor*. The internal structure of the context middleware is illustrated in figure 20.

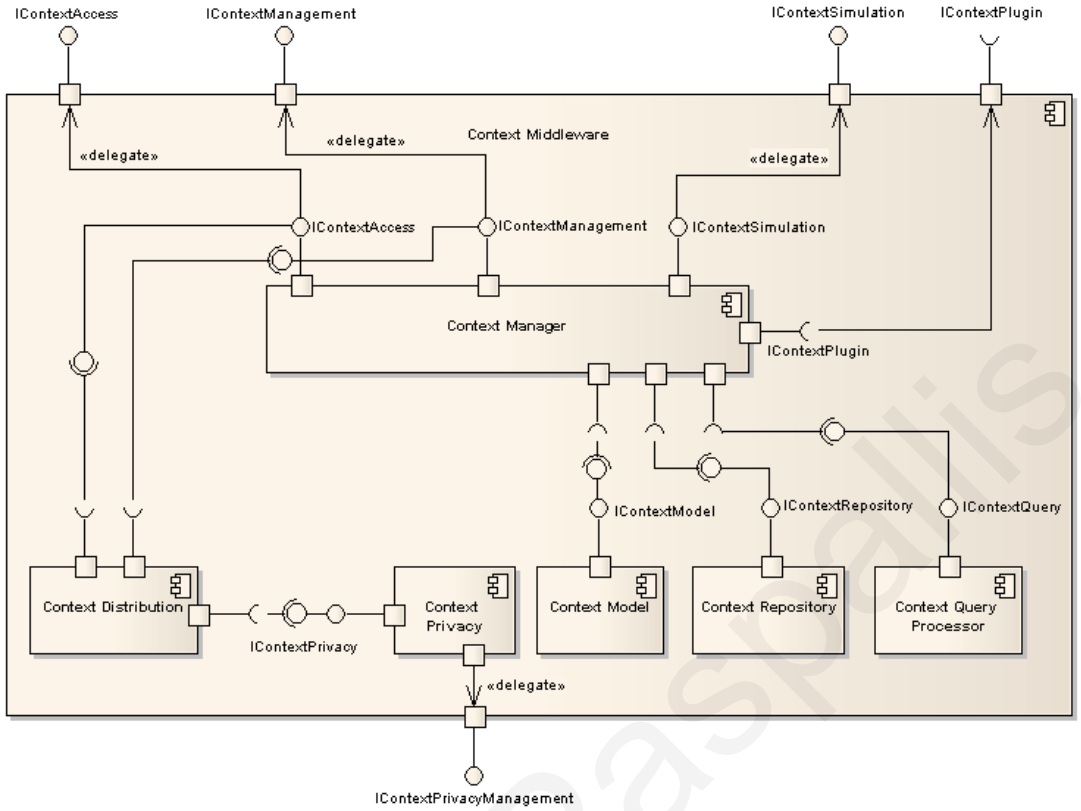

Figure 20: The modular architecture of the context middleware

5.3.1 Context manager

The context manager is the main component enabling the core functionality of the context middleware: it *receives*, *stores*, *processes* and, eventually, *forwards* context information from context providers to context consumers. It externalizes—via delegation—the three basic services of the context middleware:

- The `IContextAccess` is the basic service enabling access to context information. The queries can be either synchronous or asynchronous. In the first case, simple context types—or condition-specifying CQL queries—are used to specify the requested context data. Similarly, in the latter case, either a context type is specified so that the inquirer is notified whenever a change is sensed, or a more complex CQL query is used to specify the conditions

under which the inquirer should be notified. The implementation code of the corresponding interface is presented in listing A.1 in the appendix.

- The `IContextManagement` service primarily aims at facilitating the extension of the context middleware. For this reason, it provides the functionality for monitoring the *required* context types, as they are computed from the inquiries placed by the deployed applications, and the *provided* context types as they are reported by the installed context plug-ins. The implementation code of the corresponding interface is presented in listing A.2.

- The `IContextSimulation` is a utility service, used for testing and debugging purposes. This service allows external components to get access to the flow of context information, intercept it, and even simulate context events (with assistance from the context management service). The implementation code of the corresponding interface is presented in listing A.3.

Besides the three externally offered services, the context manager also has an externally needed service, the `IContextPlugin`. This service is used to enable the dynamic binding of context plug-ins to the middleware (see subsection 5.3.3).

Finally, the context manager depends on one optional and two mandatory internal services:

- The `IContextQuery` service provides support for processing queries expressed in CQL (see subsection 4.5.1). However, as for simple applications the simple context access API is sufficient, this service is optional allowing the formation of lighter configurations when needed (e.g., for resource-constrained devices). The implementation code of the corresponding interface is presented in listing A.4 in the appendix.

- The `IContextModel` service is mandatory and provides the functionality required for enabling IRO operations (see subsection 4.4.3) and for identifying relationships between

context entities, both at run-time. While the default implementation is realized using Ontologies, simpler object-based implementations are also possible. The implementation code of the corresponding interface is presented in listing A.5.

- The `IContextRepository` service is also mandatory and it provides the functionality needed for caching and storing historical context values. The context data are stored as objects which are serialized and can then be queried. The implementation code of the corresponding interface is presented in listing A.6.

5.3.2 Core functionality

The core functionality of the context manager includes the binding with context providers and the servicing of context clients. For these purposes, only two provided (`IContextAccess` and `IContextManagement`) and two required (`IContextModel` and `IContextRepository`) services are used.

The installed context plug-ins use the `IContextManagement` service to communicate their generated data to the middleware, which handles the task of storing and distributing them as needed. From a technical point of view, the binding of the plug-ins is achieved with OSGi's *Declarative Services* [35, 36]. The plug-ins are annotated with metadata which specify their provided and, possibly, their required context types. This information is used by the context manager which dynamically resolves the interdependencies between the plug-ins. The algorithm used by the resolution mechanism was described in subsection 5.2.3.

On the other hand, when a new application is started, it first registers its interest for specific context types using the `IContextAccess` service. This is done either by placing a query for asynchronous notification, or by synchronously querying specific context types. In the first case, the application's context needs are implicitly collected and maintained by the context manager by

analyzing the query. In the second case, however, it is impossible for the context manager to know the context needs of the deployed applications unless those are explicitly stated. For this purpose, appropriate methods are made available to the context clients so they can dynamically indicate changes in their context needs. This information is then exploited by the context manager which continuously and dynamically decides which plug-ins should be activated, as it was described in subsection 5.2.4.

Internally, the context manager utilizes the `IContextRepository` service to save the context information communicated by the providers in context events. All context information communicated by context providers is stored in the context repository, and made available for querying later on. In certain occasions, such as when the plug-ins provide the requested context types in a different representation than the one requested, the `IContextModel` service is utilized to transform the context information as needed.

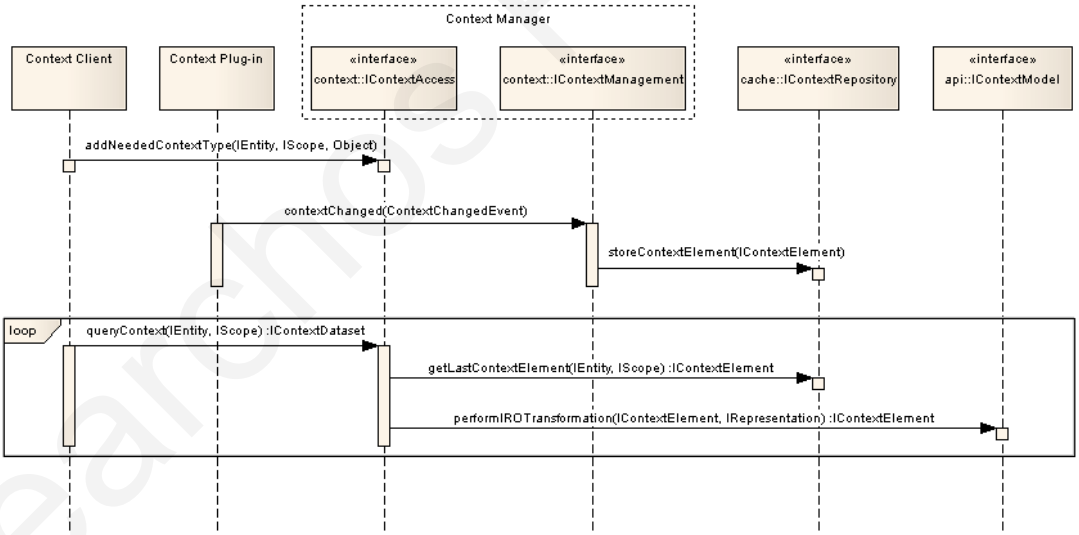

Figure 21: Sequence diagram illustrating an example of synchronous context access

A simple scenario of synchronous context access is illustrated in figure 21. In this scenario, the context client first indicates its required context types. This allows the context manager to

analyze the available context plug-ins and decide which one (or which ones) to activate (the actual activation of the plug-in is not illustrated in this sequence diagram). From that point on, the plug-in-generated context events are communicated to the context manager, which stores the data encoded in them using the `IContextRepository` service.[2] Following that, subsequent queries are served by first collecting the corresponding context information from the repository (via the `IContextRepository` service), and then by transforming its representation to the requested format (via the `IContextModel` service) as needed.

5.3.3 Architectural variability

Besides its core functionality, which is enabling the binding with context providers and context consumers, the middleware architecture also facilitates extended functionality for context visualization, simulation, testing, and distribution. Furthermore, alternative realizations can be provided for realizing the roles of the context model, the repository and query processor. An example illustrating this variability is shown in figure 22.

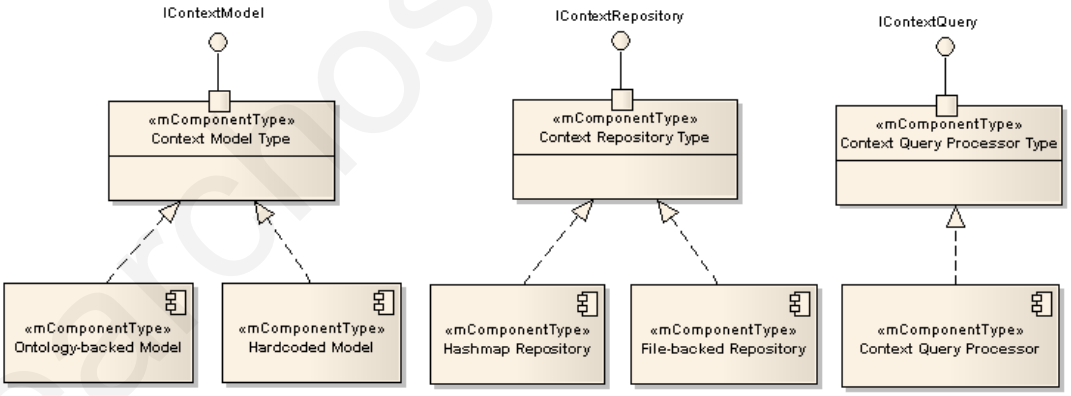

Figure 22: Variability of the context middleware architecture

This example illustrates two alternative realizations of the `IContextModel` service, two more of the `IContextRepository` service and a single realization of the `IContextQuery`

[2] In case of context clients that are registered for asynchronous notification, the events are also analyzed—possibly with the assistance of the `IContextQuery` service—and, if needed, the appropriate clients are notified with call-back methods.

service.[3] By selecting an appropriate service provider for each of these services, it is possible to formulate a middleware realization that better matches the needs of the deployment environment.

```
<?xml version="1.0"?>
<component name="context.manager" immediate="true">
    <implementation class="org.istmusic.mw.context.manager.ContextManager"/>
    <service>
        <provide interface="org.istmusic.mw.context.IContextAccess"/>
        <provide interface="org.istmusic.mw.context.IContextManagement"/>
        <provide interface="org.istmusic.mw.context.IContextSimulation"/>
     </service>
    <reference name="context.plugin"
        interface="org.istmusic.mw.context.plugins.IContextPlugin"
        bind="installPlugin"
        unbind="uninstallPlugin"
        cardinality="0..n"
        policy="dynamic"/>
    <reference name="context.model"
        interface="org.istmusic.mw.context.model.IContextModel"
        cardinality="1..1"
        bind="setContextModelService"
        unbind="unsetContextModelService"
        policy="dynamic"/>
    <reference name="context.repository"
        interface="org.istmusic.mw.context.cache.IContextRepository"
        cardinality="1..1"
        bind="setContextRepositoryService"
        unbind="unsetContextRepositoryService"
        policy="dynamic"/>
    <reference name="context.query"
        interface="org.istmusic.mw.context.cqp.IContextQuery"
        cardinality="0..1"
        bind="setContextQueryService"
        unbind="unsetContextQueryService"
        policy="dynamic"/>
</component>
```

Listing 5.3: Declarative services configuration file for the context manager

The service descriptor in listing 5.3 depicts three offered services (i.e., context access, context management and context simulation) and four required services (i.e., the externally bound context

[3] The `IContextQuery` service is not intended to have multiple realizations. Rather, the complete service provider might be omitted if CQL-support is not required, or device-related constraints demand it.

plug-ins, and the internally bound context model, repository and query services). It should be noted that the cardinality properties defined in this descriptor indicate that *any* number of context plug-ins is allowed to be connected, while *exactly one* realization of the context model and context repository services are allowed. The context query realization has an *optional, up-to-one* dependency.

The service descriptor illustrated in listing 5.3 is used by the *Declarative Services* system to automatically manage the bindings of the services as needed. While the presented architecture can be reconfigured even during run-time, the design is not intended for this purpose. Besides the context plug-ins, which are specifically designed to be dynamically added or removed at run-time, the context model, repository and query service providers are not intended to change at run-time.

5.4 Context distribution

Context distribution is the process of communicating context information between devices with the purpose of sharing it. In some cases context distribution is simply useful as it enables either richer context information to reach a wider set of devices or more cost efficient context sensing, or both. For example, when a mobile device is connected with a car-embedded computer to receive location information from it rather than using its own GPS sensor, it is effectively provided with both richer (i.e., more accurate) and more resource efficient context data (i.e., assuming the resources required for distribution are less than those needed to run its own GPS sensor). However, in some other cases, context distribution is a necessity as many applications make use of context information which is natively distributed (e.g., monitoring the status or the location of your friends).

To enable context distribution, two main challenges need to be addressed. First, the exchanged data must be modeled appropriately to overcome the barriers of semantic consistency and functional interoperability. These barriers were discussed in subsection 4.4.4, where it was also presented how the proposed context model overcomes them. The second challenge refers to the actual communication of the context information. For instance, how the peers are discovered and how the data are communicated. Naturally, this challenge also incurs security and privacy concerns.

5.4.1 Distributed context providers and context consumers

From a centralized point-of-view, the context manager keeps track of all the available context types (as they are reported by the deployed plug-ins) along with all the needed context types (as they are reported by context clients and context reasoner plug-ins). This information is used to dynamically activate the appropriate plug-ins as needed. However, when the information is also expected to be shared across different context spaces (see figure 7), then the context manager needs to also take into consideration the context types that are offered by distributed providers (so that it can deactivate local plug-ins if needed) and also the context types that are needed by them (so that it can activate the corresponding plug-ins).

In the proposed architecture, context distribution is achieved by viewing the distributed context spaces as separate context providers and context consumers (see figure 7). Appropriate distribution mechanisms are implemented as separate components (e.g., independent OSGi bundles). Such mechanisms utilize the `IContextAccess` and`IContextManagement` services in order to realize special context providers and context consumers, in effect extending the context space to a wider one, covering multiple instances of the middleware. Naturally, these mechanisms are expected to also deal with additional challenges, such as security and privacy (i.e., making sure that only authorized clients are provided with access to specific parts of the context data).

Similar to the centralized management of context providers and context consumers, the distributed context management system provides no guarantee of delivering the required information (i.e., the distributed context providers and consumers remain loosely coupled). Rather, the context manager follows a best-effort approach where the local plug-ins are activated, when possible, to accommodate distributed context needs. In cases where the context information is not available though (e.g., because the corresponding sensor is disconnected), the distributed context requests are never served. This is, of course, an unavoidable outcome for scenarios where none of the participating peers provides the requested context type.

5.4.2 Conceptual model of context distribution manager

A context distribution manager is an independent component (packaged as an OSGi bundle in this case) which can be used to extend the capabilities of the centralized context middleware to also cover context distribution. For this purpose, a realization of the context distribution manager should provide two fundamental functionalities: *peer discovery and grouping* and *context communication*.

Peer discovery is the task of discovering nearby (publicly available) and remote (authorization-based) context sources. Clearly, only a subset of context distribution peers should be discoverable in order to keep scalability and privacy issues contained. Both directory-based and ad-hoc group forming approaches are possible. Furthermore, the group-formation itself can be based on additional properties (such as privacy and additional context attributes as described in [84]).

Context communication is the task of encoding the context information and transferring it from the source to the intended target, where it is decoded. Based on the intended application, the actual communication can be based on ad-hoc or infrastructure-based networks.

The challenges of peer discovery and group formation as well as that of context communication have been addressed in various ways, as described below.

5.4.2.1 Ad-hoc based context distribution manager

An ad-hoc based design of a context distribution manager was proposed and implemented as described in [104]. In this design, groups were implicitly formed within a wireless *Local Area Network* (LAN) using a broadcast mechanism. This kind of opportunistic formation of temporary groups is intended for sharing publicly available context information only (such as current location, temperature, etc).

This approach has the important advantage of assigning higher importance to local context and consequently promoting localized scalability [122]. Arguably, it is more likely that two neighboring nodes will share a common interest on the same context types as opposed to nodes at remote locations. This is true in most pervasive computing applications which aim to utilize the infrastructure embedded in the surrounding environment. More specifically, it is more likely that an application would be more interested in the temperature information provided by nearby nodes (and thus residing in the same environment) as opposed to the temperature information provided by distant nodes. Furthermore, localized scalability is achieved by preferring local context providers (and, respectively, local context consumers) for sharing context information with. In this approach, the use of infrastructure links is avoided as most of the communication is carried out over local (i.e., direct) network links, further contributing towards scalability.

5.4.2.2 Peer-to-peer based context distribution manager

Beyond the ad-hoc based approach, peer-to-peer networks can also be utilized to enable context distribution. By avoiding a centralized server, a more reliable and resilient mechanism can be realized for context distribution.

In [75, 76], it is argued that peer-to-peer systems can facilitate context distribution. The proposed peer-to-peer based architecture reuses the core services of context access and context management (see subsection 5.3.2) while also introducing three distribution-specific services: *discovery*, *group formation* and *communication*. The first one is used for realizing service discovery in the peer-to-peer context, the group formation service uses the discovered services and a predefined policy to form peer groups and, finally, the communication service is used for enabling context sharing among nodes in the same group. The group policy also allows the formation of multiple peer groups, with different sharing privileges. For example, a *protected* group can be used for sharing "*precise location*" among colleagues, and an *unprotected* group can be used to share more coarse context data like "*current city*".

A prototype of this design was partially implemented on top of JXTA [5], a standard technology that allows connected devices on the network to collaborate in a peer-to-peer manner. By building on the peer-to-peer architecture, the context distribution manager satisfies the heterogeneity, scalability and robustness requirements, as they are automatically inherited from the JXTA technology [76].

In a separate work, Devlic et al realized a similar peer-to-peer based context distribution manager. Unlike the JXTA based approach however, this approach is based on the *Session Initiation Protocol* (SIP), which is used for both the discovery of participating nodes and for the communication of context information [43, 21]. This architecture includes a context privacy manager, allowing for customized configuration of the privacy protocol. The SIP-based architecture has been implemented in the scope of the IST-MUSIC project [6], and has been integrated with the context manager as it is illustrated in the middleware architecture shown in figure 20.

5.5 Context visualization and simulation

Context visualization and simulation are both important features in context-awareness enabling frameworks, as they allow the developers to better monitor, test and debug their applications and the frameworks themselves.

Context visualization is enabled with the use of the context access and context simulation services provided by the context manager (see subsection 5.3.1). The `IContextAccess` is used to enable explicit requests of context information, just like it is used in context-aware applications. The `IContextSimulation` allows for the interception of *all* context events, so that a single client can be used to visualize all context data as reported by the context providers. The intercepted events can be eventually routed back to the system—for storage and delivery to the context clients—or they can be suppressed.

Context simulation is enabled with the use of the context management and context simulation services, also provided by the context manager. While the `IContextSimulation` service allows for the interception of all context events, the `IContextManagement` service provides the functionality for simulating any valid context value. Combining the functionality offered by these two services, an external component can be realized for intercepting and blocking all *real* context events (as they are reported by the installed plug-ins) and for producing simulated context values (e.g., as defined in a script-file).

An incarnation of such a visualization and simulation tool was implemented and is illustrated in figure 23. The context viewer component (also packaged as a separate OSGi bundle) enables testing and debugging of both the middleware architecture and context-aware applications. The first two tabs allow for monitoring the provided and required context data as reported by the context manager. The first allows, for instance, to check if the corresponding context provider

plug-ins have been installed and started correctly and the second allows for viewing the context types requested by the context clients. Furthermore, the user can select to add a custom context listener for any context type, which allows for testing the plug-in activation mechanism. The custom context listeners appear in the third tab, along with any events they receive. Finally, the context simulation tab appears in the fourth tab. A specialized script is used to indicate when a message should be logged, a context event created and how long to wait before proceeding to the next event.

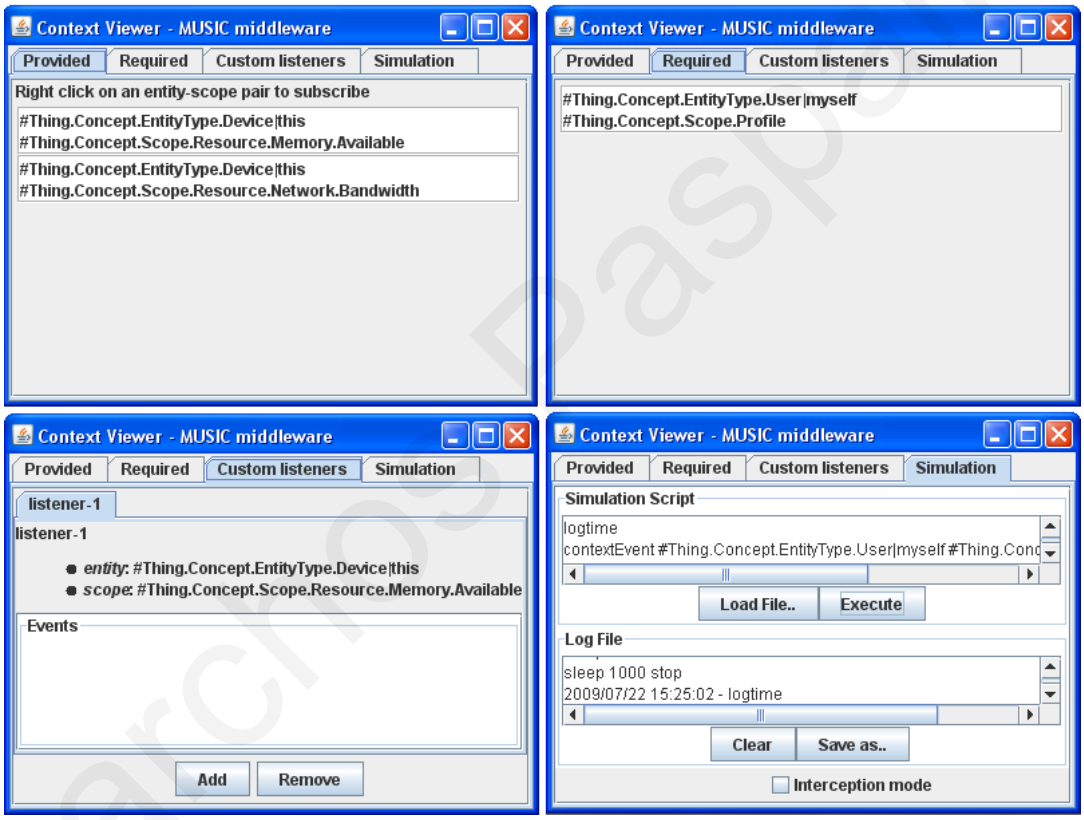

Figure 23: The context viewer component used to visualize and simulate context events in the context middleware architecture

5.6 Implementation issues

The implementation of the context middleware architecture is based on *OSGi Alliance*'s Java-based service platform (commonly referred to as OSGi). The adoption of a standardized framework results in a smoother learning curve for new adopters and was selected to make the framework more appealing to the developers.

While multiple implementations of the OSGi specification are already available, the context middleware architecture was implemented using Eclipse's Equinox.[4] The selection of Equinox was mainly due to factors like license, size, platform, tool, and community support. However, as the implementation code is strictly OSGi R4 compliant, the middleware works correctly on compatible implementations (so far, beyond Equinox, the middleware has been successfully tested also with Knopflerfish).[5]

Besides the requirements listed in section 5.1.1, the design and implementation of the context middleware was, naturally, also based on the experiences gained from MADAM [7], the MUSIC project's predecessor [6]. These experiences were documented by the author in [104] and include a variety of recommendations regarding performance, extensibility, reuse and modularity.

For instance, a main drawback of the original implementation [104] concerned its performance. It was since found that the more plug-ins were installed, the less efficient the system became, posing a scalability issue. The limitation was found to be related to the burden that multiple threads placed on lightweight devices. To overcome this limitation, a custom scheduler was developed to drive all the installed plug-ins from a single thread-pool. Rather than requiring multiple concurrent threads, the scheduler allows interested clients to allocate either a one-time or a periodic invocation for a predefined portion of code. It is then up to the scheduler to handle

[4] Eclipse Equinox, *http://www.eclipse.org/equinox*
[5] Knopflerfish OSGi, *http://www.knopflerfish.org*

these requests using one or more threads as needed. In the current implementation, a single-threaded mechanism was tested, showing significant performance improvement. While the same device's responsiveness was significantly degraded with as few as three plug-ins in the original implementation, the scheduler has allowed the deployment of twice as many plug-ins with nearly un-noticeable performance degradation.

Another improvement compared to the original codebase is the adoption of an event-based approach in the context middleware. In this regard, the context manager encapsulates an *event queue* which makes it easier to handle events, and optimize their execution. Typical examples of such events include the installation and un-installation of context plug-ins, changes to the set of needed context types and, naturally, context change events as they are initially generated and communicated by context plug-ins. Periodically handling these events as they are grouped in a queue has important advantages: It only requires periodic attention and thus does not exhaust the processing resources of resource-constrained devices (the context manager actually uses the same scheduler mechanism used to drive the plug-ins and described in the previous paragraph). Additionally, it allows for streamlining and optimizing the execution of events in the pipeline, by handling them in batches when possible. For instance, when multiple consecutive events in the queue indicate the need for *resolving* or *activating* the plug-ins, those are merged and processed as one. Even more drastically, when two events indicate the need for *resolving* and *activating* the plug-ins, those are also merged. This is justified because handling *resolving* events always causes the invocation of both the resolution and the activation mechanisms, which renders the handling of subsequent *activating* events obsolete.

Finally, since the middleware aims for mobile and pervasive computing environments, the codebase was kept as small and as portable as possible. As of version 0.4.0 of the middleware, the context management system bundle had a size of 162Kb (this includes the sub-components of the

context manager, context cache and CQP). Furthermore, to keep the code as portable as possible, the middleware was implemented using J2SE version 1.3, which is supported by many JVMs built for mobile devices, like NSI's CrEme and the open-source Phone-ME. Finally, it should be noted that the complete MUSIC middleware stack, including the context middleware and core plug-ins, have also been ported to the Android platform.[6]

5.7 Discussion

The context middleware architecture presented in this chapter serves as both a stand-alone framework for enabling context-aware applications while, at the same time realizing a crucial component of the MUSIC middleware [6]. Since the latter features autonomic self-adaptation, it incorporates an *adaptation reasoning* module which is, naturally, a main user of context information. In this respect, the mission of the context middleware architecture can be summarized as *collecting*, *organizing*, and *storing* relevant context information, and *making it accessible* to context consumers, both at the middleware and the application levels.

In more detail, this middleware architecture facilitates three types of scenarios:

- simple context presentation applications that need to be easily and quickly prototyped, requiring only minimal interfacing with the middleware;
- more complex context-aware applications which need to be cost-efficiently developed, by reusing existing code (e.g., context providers) whenever possible and by adding new code only where needed; and
- integration of the middleware architecture with other components which require access to context for enabling advanced autonomic behavior (e.g., self-adaptiveness [54]).

[6] It should be noted that the porting to Android is—at the moment—limited to running the OSGi and the MUSIC middleware as a single Android application, which is a significant limitation. The current results of this porting are publicly available at the project's website [6].

The first point refers to applications which simply access context information with the purpose of displaying it, either directly or indirectly. For example, a mapping application accesses *location* information with the purpose of either displaying the coordinates or centering its map.

Other applications might require more complex functionality, such as inferring the user activity based on their location and agenda. In this case, the developers should be supported to reuse existing code (e.g., for accessing the raw *location* and *agenda* information) as well as define their own context-awareness logic (e.g., for inferring the *user activity*).

However, more complex context-aware logic typically requires advanced mechanisms, enabling autonomic behavior. Three well-known approaches from the field of *Artificial Intelligence* (AI) are *action-based*, *goal-based* and *utility function-based* [135]. In order to maintain the domain of the middleware architecture as flexible as possible, all three approaches are supported.

In the following paragraphs it is argued that the proposed middleware architecture accomplishes this mission while at the same time it introduces some features which improve productivity at design-time, and resource consumption at run-time. In chapter 7 it is further argued that the proposed middleware architecture, in combination with the development methodology presented in chapter 4, satisfy the majority of the requirements identified in the literature (see table 1).

5.7.1 Related work

Unlike approaches which aim at realizing the full extend of their context-aware logic inside the applications (e.g., CML [68, 70] and COSMOS [41, 117]), or approaches that focus on specific domains like ad-hoc network topologies (e.g., EgoSpaces [79]), the presented middleware architecture aims at providing a flexible and customizable solution, where the developers can customize or extend it as needed (similar to the approaches taken by the Context Toolkit [44, 46] and Aura's Context Information Service [78, 129]).

The COSMOS approach is a component-based framework for managing context information in ubiquitous, context-aware applications (see subsection 3.3.14). The basic structuring concept in COSMOS is the *node*, which can be thought of as similar to a context sensor or a context reasoner plug-in. Unlike plug-ins though, the nodes correspond to context types and they are used to define logic hierarchies which are in turn used to define the context-aware behavior. On the other hand, the context plug-ins defined in our approach are software entities that can be installed or un-installed and also be activated or deactivated.

By providing a uniform abstraction of context information—the context node—COSMOS supports the composition of context information from low-level sensors to high-level policies. At the lowest level, context nodes reify hardware capabilities, software resources, or embedded sensors. At a higher level, context nodes reuse or develop composition operators to infer advanced context information. Evidently, this is similar to the context sensing and reasoning hierarchy proposed in this thesis (see subsection 4.2.4). However, the use of node-based context-aware logic is limited to run-time access by single applications only, and does not facilitate activation or deactivation of the actual hardware sensors. In contrast, the proposed pluggable architecture facilitates sharing of context information among concurrent applications. Furthermore, it allows for automatic activation and deactivation of the plug-in components which is an important advantage for applications deployed on resource-constrained devices.

As stated in [117], the core motivation of COSMOS is to isolate context management policies from applications and to enforce their reuse. Because context policies are themselves reflected as context nodes, they can be reused in different contexts. However, as the nodes in COSMOS correspond to finer-grained concepts (such as operators), reusing them can be quite elaborate. In contrast, the context plug-ins are well-defined components, as per the OSGi specification, with added context-specific annotations making them easy to reuse.

The CML is an extensive software engineering framework for enabling context-aware pervasive computing (see subsection 3.3.11). The authors present their view of context awareness as a technique which enables pervasive computing by allowing applications and systems to act autonomously on behalf of users [67]. Henricksen and Indulska argue that their framework achieves this goal while also addressing three basic challenges that they identified (also see section 3.1): analysis of application's context requirements, acquisition and management of the relevant context data and, finally, design and implementation of suitable context-aware behavior .

Unlike the current state-of-the-art, the approach proposed in this thesis offers a methodology and a middleware architecture for developing, deploying and maintaining context-aware applications. This methodology separates the design and the development of context producers (i.e., sensor and reasoner plug-ins) from that of context consumers (i.e., context-aware applications). Furthermore, the underlying middleware handles a number of common functionalities such as lifecycle support for the plug-ins, context aggregation, context storing and a rich query language. It also facilitates the development of autonomous, context-aware applications in which the decisions are taken using an elaborate hierarchical process.

One of the most important advantages of the presented architecture is its modularity, which facilitates the formation of various instantiations matching the capabilities and the needs of the deployment environment. Also unlike the related work, the proposed architecture allows for dynamically added/removed context sensors and, additionally, it facilitates their dynamic activation. As it is experimentally shown in the following subsection, this ability provides significant resource optimization.

5.7.2 Experimental evaluation of resource optimization

In order to evaluate the effectiveness of the proposed resolution and activation algorithms, a prototype solution was implemented based on the middleware implementation and a set of context plug-ins (some of them real and some of them simulated). The middleware is used to deploy a context-aware application executing a scenario where context changes dynamically. While the scenario progresses, the performance of the algorithms is measured by monitoring the resource consumption (battery and memory usage) in the device. The scenario was initially executed with the automatic activation mechanism enabled, which resulted to the dynamic activation of the plug-ins as needed. Then, the scenario was repeated with the mechanism disabled, which resulted in all the installed plug-ins always being active.

The evaluation focused on *Personal Digital Assistant* (PDA) devices because resource consumption is more critical for this type of resource-constrained devices. An HP iPAQ 6910 handheld, running Windows Mobile 5.0 was used in the experiments. This device was chosen as it is equipped with both WiFi and Bluetooth adapters, a GPS receiver, and GSM phone capabilities. This equipment allowed the implementation and test of a rich set of real plug-ins in addition to a set of simulated ones. The exact approach is described in the following subsections.

5.7.2.1 Implementation of the context plug-ins and experimental setup

In order to implement the simulated scenario, a set of real and simulated context plug-ins was defined, as listed in table 3. These context plug-ins were implemented using pure Java [22], but for some operations (e.g., Bluetooth, Wi-Fi, phone activation and screen brightness adjustment), low-level access to the underlying operating system was required. This was achieved with the use of the *Java Native Interface* (JNI) technology which allows the invocation of methods contained

in native libraries from inside Java code. A native library was implemented using the C++ language, to wrap the calls to the required native methods and link them to the corresponding Java code, according to the JNI specification. Also, since the focus was on PDAs, a suitable run-time environment was selected to execute the Java bytecode. For the purposes of this experiment, the CrE-ME 4.0 *Java Virtual Machine* (JVM) from NsiCom Ltd was selected, which is based on the JDK 1.3.1 specification and supports the J2ME CDC Personal Profile.

Table 3: The context plug-ins used in the experimental evaluation

Plug-in name	**Type**	**Description**
Bluetooth sensor	Real	Allows to switch on and off the Bluetooth adapter
WiFi sensor	Real	Allows to switch on and off the WiFi adapter
GSM sensor	Real	Allows to switch on and off the phone functionality
Location sensor	Real	Allows to switch on and off the GPS receiver of the device and to retrieve information on the current geographical position
RFID sensor	Simulated	Simulates the detection of an RFID tag
Light sensor	Simulated	Simulates the detection and measurement of the ambient light of the environment where the device resides
Weather reasoner	Simulated	Based on the location info (e.g., provided by the Location sensor) and on Internet access (provided by the WiFi adapter and controlled by the WiFi sensor), it simulates the invocation of a web service providing weather forecast for the given location

5.7.2.2 The tourist scenario

The selected scenario consists of a sequence of actions performed by a tourist while visiting a city. The tourist uses a hand-held PDA device, which hosts the middleware described earlier, along with an application which defines a set of context needs and reactions to context changes. The reactions consist of actions that adapt the device configuration with the objective of optimizing the resource utilization and maximizing the user utility, in the new context.

The scenario is organized in phases called *scenes*, all of which have a predefined duration. The scene descriptions and the context changes that occur in the environment for each one of them are listed in table 4. Moreover, the "description" column explains how the application adapts to changes when the plug-in mechanism is enabled. In order to evaluate the effectiveness of the proposed mechanism, the same scenario was repeated having the plug-in mechanism disabled. In this case, all the components needed by the application—i.e., the GPS, Bluetooth, and WiFi adapters, the RFID reader, the light sensor, and the phone—were always switched on and the screen brightness was set to max.

5.7.2.3 The simulation application

The simulation of the scenario was driven by an application that reproduced the context changes and the user actions according to the scenes described below. When the application was launched with the context plug-in mechanism enabled, all the steps involving plug-in resolution and activation were performed, and the application reacted to the context changes by performing the corresponding adaptations. On the other hand, when the context plug-in mechanism was disabled, all needed low-level resources were switched on—at start-up—and no adaptation was performed. These steps are described in table 4.

In addition to the tasks described above, the simulation application performs the measurements of the memory and battery consumption in a separate thread. It does so by polling the underlying operating system at predefined intervals (i.e., every 5 seconds) to retrieve the resources status. In this way, the remaining battery charge, the actual memory usage (in terms of bytes and memory load as a percentage), the timestamp of the measurement and the corresponding scene name are all saved in a file to be later analyzed.

Table 4: The scenes of the experimental evaluation

Scene	Duration	Description
0	95 sec	The tourist leaves the hotel for a tour of the city. He starts the guide application on the PDA. As the application requires RFID, GSM and light sensors, the associated plug-ins are resolved and activated.
1	300 sec	The tourist is walking in the street and the weather is sunny. As the ambient light changes from normal to intense, the screen brightness is automatically set to maximum (better contrast).
2	600 sec	The tourist reaches an area providing a WiFi network. He now walks in a shady alley and ask to be notified periodically of weather forecast. As the ambient light changes from intense to normal, the screen brightness is set back to medium (better contrast and lower power consumption). At the same time, the application requires weather information and the weather plug-in is thus resolved and activated.
3	30 sec	The tourist enters a metro station and the RFID reader detects a tag providing the following information about the environment: "Station DUOMO", no WiFi, no Bluetooth kiosk, and no GSM coverage. Thus, the WiFi, GPS, and Phone adapters are switched off and the weather plug-in is deactivated while the Bluetooth adapter is switched on.
4	1200 sec	The tourist waits in the metro station. As the light sensor detects a change due to the station lights, the screen brightness is set to low in order to save battery.
5	30 sec	The tourist gets off the couch and walks towards the exit. The RFID reader detects a tag providing the following information: "Station LORETO", Wi-Fi and GSM available, no Bluetooth kiosk. Thus, the Wi-Fi and GPS adapters are turned on, the weather plug-in is activated and the phone is switched on.
6	30 sec	Now the weather outside turns to cloudy. As the light sensor detects this change, the screen brightness is set to medium.
7	600 sec	The tourist walks around the city guided by its GPS, receiving weather forecasts on a display whose brightness is set to medium and having its GSM turned on.
8	-	The tourist reaches the hotel and quits the application. As a result, all the context plug-ins required by the application are now deactivated and release their allocated resources.

5.7.2.4 Experimental results

As discussed earlier, the goal of this experiment was to evaluate the effectiveness of the context middleware and its dynamic activation mechanism. In this regard, the experiments were repeated twice: once with the plug-in mechanism enabled and a second one where it was disabled. All the experiments were initiated with the battery fully charged and without other applications running, in order to keep the execution environment as consistent as possible during the two runs.

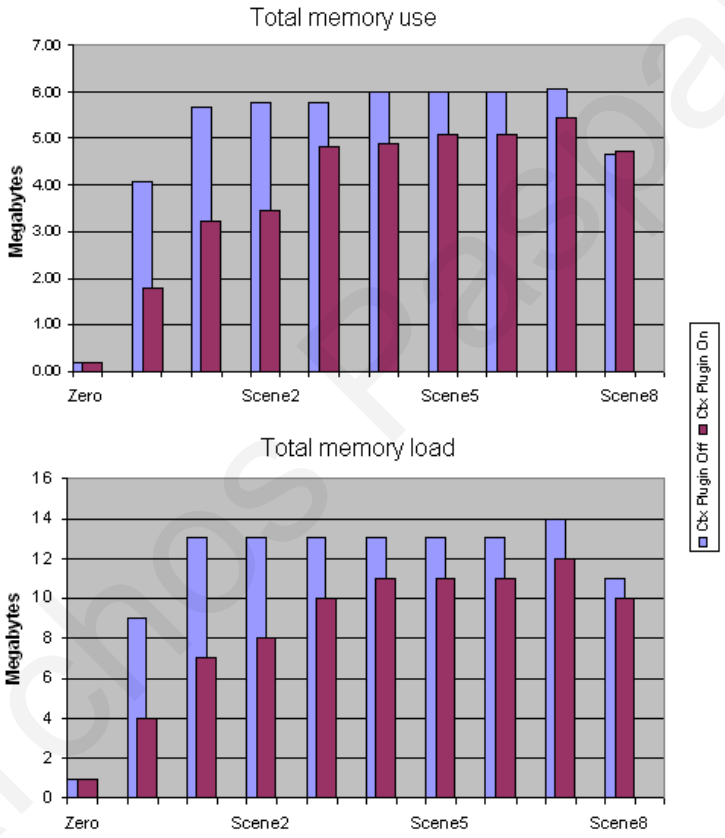

Figure 24: Total memory use and total memory load comparison

The graphs of figure 24 depict the trends in memory use (in terms of absolute memory used in MBytes and also in terms of system memory load as a percentage). These trends are presented as a function of the scenario scenes. As shown in the graphs, the execution with the context

plug-in mechanism enabled was more memory efficient, in almost all the scenes. Table 5 shows a summary of the numerical values measured during the experiment. Enabling the context plug-in mechanism has saved an average of 21.29% of the total memory used, compared to when the mechanism was disabled. In terms of memory load, the average gain was 17.44%.

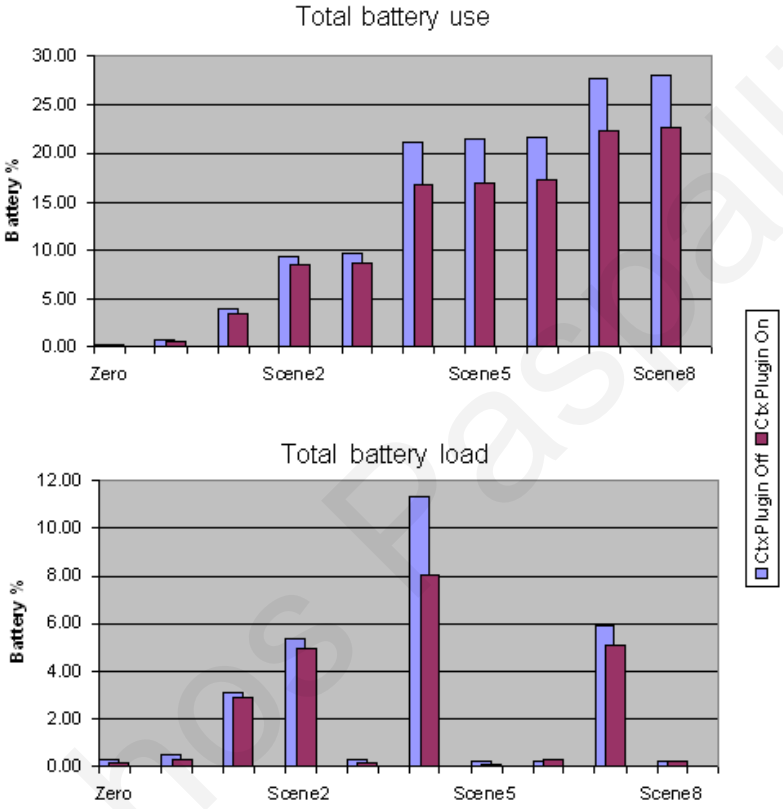

Figure 25: Total battery use and battery use per scene comparison

The graphs of figure 25 depict the battery load during the execution of each step of the scenario. The first graph shows the trend of the total battery consumption, while the latter illustrates the relative battery usage for each scene. The execution with the context plug-in mechanism enabled was proven to be more energy efficient in all the scenes.

As depicted in table 5, the memory utilization was improved by 21.29%. The TMU column corresponds to the *Total Memory Use* and is measured in Megabytes (MB), while the TML column is the *Total Memory Load* and is measured as a percentage. Furthermore, the TBU column corresponds to the *Total Battery Use* and is measured as a percentage and finally the BPUPS is the *Battery Percent Used Per Scene* and is also measured as a percentage.

Table 5: Memory and battery measurement results during the experimentation

	TMU (MB)			**TML** (%)			**TBU** (%)			**BPUPS** (%)		
Scene	Off	On	Δ%	Off	On	Δ%	Off	On	Δ%	Off	On	Δ%
Zero	0.19	0.19	0.00	1	1	0.00	0.30	0.19	37.84	0.30	0.19	37.8
Init	4.07	1.80	55.8	8	4	50	0.83	0.54	34.57	0.48	0.32	32.2
S 1	5.67	3.21	43.4	12	7	41.7	4.00	3.43	14.27	3.13	2.86	8.59
S 2	5.74	3.45	39.8	12	8	33.3	9.41	8.40	10.80	5.38	4.92	8.38
S 3	5.74	4.82	16.1	12	10	16.7	9.75	8.63	11.49	0.29	0.19	34.1
S 4	5.99	4.88	18.5	12	11	8.33	21.07	16.73	20.62	11.28	8.06	28.6
S 5	5.99	5.06	15.5	12	11	8.33	21.38	16.87	21.09	0.27	0.10	64.6
S 6	5.99	5.06	15.5	12	11	8.33	21.69	17.22	20.59	0.27	0.30	-12.8
S 7	6.06	5.45	9.99	13	12	7.69	27.64	22.33	19.22	5.92	5.07	14.4
S 8	4.63	4.71	-1.7	10	10	0	27.96	22.65	18.98	0.25	0.27	-9.1
	Avg Δ%: 21.29			Avg Δ%: 17.44			Avg Δ%: 20.95			Avg Δ%: 20.68		

Furthermore, the activation of the plug-in mechanism has saved 20.95% of the total battery consumption. In absolute terms, for the used device, it means that 6% of the total battery capacity was saved. Similar results were also reflected in the relative battery usage per scene, where the average gain was 20.68%.

5.8 Conclusions and future work

This chapter presented a middleware architecture, which was designed to support the development and deployment of context-aware applications. The architecture was designed based on the concepts and the methodology presented in chapter 4, aiming to facilitate their main objectives:

software reusability. At the same time, the design of the middleware aimed for a pluggable and modular architecture, satisfying the requirements identified in section 3.2.

The presented middleware architecture has been implemented and is used in the MUSIC middleware [6], where it serves as the context manager component. The architecture itself evolves the MADAM context system's architecture, which was also pluggable but not modular and featured a primitive context model among other limitations [104]. While the middleware has been extensively used in the pilot applications of the MUSIC project [6], it has also been experimentally evaluated in the context of an undergraduate course (see section 7.3).

While the middleware already appears as a powerful and flexible solution for context-aware applications, more features and design changes are planned to further enrich its functionality. For instance, the activation mechanism described in subsection 5.2.4 is being revised to allow for more intelligent activation of the plug-ins. While the current mechanism starts or stops all the plug-ins offering a needed context type, new methods are investigated to allow a more granular approach. In this, the plug-ins will be checked not only against their provided and required context types, but also against their profiled resource consumption (e.g., battery, network, memory and CPU use) so that only a subset of the plug-ins is activated, thus optimizing their resource consumption. Furthermore, additional functionality is being designed to transform the context plug-ins into self-adaptive components. This would allow, for instance, to adapt the polling interval of a plug-in, or the fidelity of its generated context info, thus optimizing its resource consumption.

Chapter 6

Model-driven development of context plug-ins

As argued in chapter 4, the development of context-aware applications can be eased by structuring them as context providing and context consuming components. The architecture and the properties of the context plug-ins were presented in chapter 5, where it was also shown that this approach can result in better resource utilization. This chapter revisits the context plug-ins and proposes a model-driven development approach for their construction. This approach has the advantage of providing an easier, graphical environment-based approach for the development of the plug-ins while at the same time further facilitating software reuse in terms of more elementary data-structures and context processing operators.

6.1 Introduction

In *Model-Driven Development* (MDD), newly defined domain-specific modeling languages, or extensions to existing languages, are generally used to define models of the desired system at an abstract and platform-independent level. The defined *Platform Independent Models* (PIMs) are automatically transformed to *Platform Specific Models* (PSMs) and platform-specific code in one or more steps. For this purpose, a tool chain is typically incorporated into the corresponding development framework. Application developers are not confronted with implementation details

and thus they can concentrate on the high-level view of the software to be developed. In this way, many conceptual and implementation errors can be avoided upfront.

The intention of the proposed approach is to complement the development methodology by exploiting the benefits of MDD for the creation of context plug-ins. These plug-ins are designed to be used as reusable components on the middleware architecture presented in chapter 5, for the formation of complete context-aware applications. Starting from a pure conceptual model that introduces the main concepts and their relationships, a new UML Profile is presented. This profile is used to facilitate the modeling of context plug-ins at an abstract level. Then, a corresponding tool chain—which is also described—is used to realize the transformation of the model into code.

6.2 Conceptual model

In MDD, a conceptual model is used to describe the basic entities used in the development approach, along with their relationships. The relationships between the entities include extension (inheritance) and aggregation. The conceptual model presented here aims at specifying the main concepts required to implement context plug-ins. As such, it heavily builds on the context model and the context query language presented earlier in chapter 4.

The UML diagram in figure 26 shows the conceptual model comprising the main entities and their relationships. These entities are the *context plug-ins*, the *operators*, the *Data Management Containers* (DMCs) and the *connectors*.

A context plug-in represents the architectural element that is deployed on the context middleware. Externally it is mainly characterized by its provided and required context types. The plug-ins are interfaced with the context system (i.e., to identify their required and provided context types) by specifying basic data structures for receiving and sending context information. These data structures are modeled using—possibly multiple—Input DMCs and a single Output DMC.

The trigger type for these DMCs can be time-based (OnTime) or event-based (OnChange). The *trigger* value defines the trigger type through an appropriate parameterization (i.e., via a trigger interval or by specifying the event that fires the trigger).

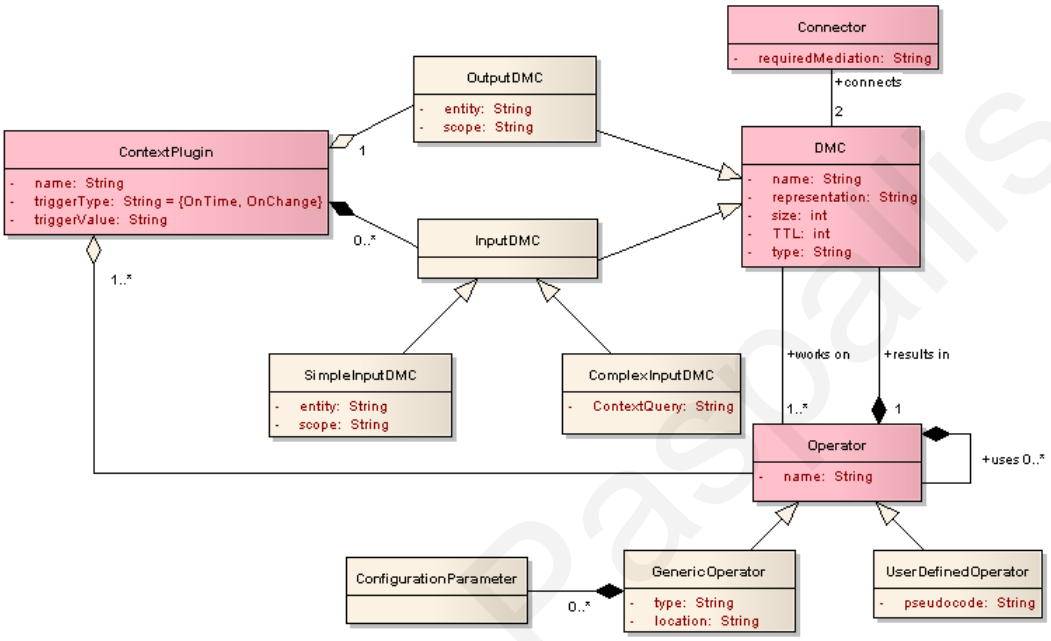

Figure 26: Conceptual model of context plug-ins.

In the same way as in [24], a *DMC* typically represents a common data structure (e.g., a simple variable, array, list or queue). The DMCs are simply used for caching information without implementing any further functionality. A DMC is characterized by its type (e.g., variable, array, queue), its size (i.e., number of elements to be cached) and its *time-to-live* (TTL) attribute encoding the validity period of its contents. Additionally, DMCs also include a *representation* attribute. This attribute refers to the semantic representation concept of the model (see subsection 4.4.2) and thus defines the data-type of the stored elements.

The *Input DMC* and *Output DMC* entities are specializations of the DMC entity, and are incorporated by a context plug-in to interact with the context system by defining its required and

provided context information. An Output DMC characterizes the provided information by specifying the scope of the provided information, as well as its corresponding entity and representation (inherited from DMC). An Input DMC specifies the required context information and provides two further specializations: First, a *SimpleInputDMC* which is used to model the required information simply through the corresponding entity, scope and desired representation of the information (see subsection 4.4.2). However, as the underlying context middleware also provides elaborate context access through the CQL, it is also allowed to express the required context information through an associated *query*. For this purpose, the *ComplexInputDMC* is introduced. As a context query allows refining the requested information by specifying conditions on values as well as on metadata (e.g., a timestamp), this also enables the developers to deal with aspects like data freshness.

Operators are conceptual entities that provide well-defined functionality by performing specialized calculations on the data cached in one or several—input—DMCs. The results are again stored in a DMC. Operators are distinguished between *generic* and *user-defined*. Generic operators are predefined (i.e., pre-implemented) operators that can be reused in many different context plug-ins and usually they are provided as part of a library. A generic operator is associated with a number of configuration parameters which allow fine-tuning of its behavior. Examples of generic operators are provided in section 6.3. In contrast to generic ones, user-defined operators define more specific functionalities which usually renders them useful only within the domain of a single context plug-in. The corresponding function body can be specified in pseudo-code in special place-holders, which are included in the automatically generated source code as comments.

While operators are the main computational artifacts and are used to define the internal functionality of context plug-ins, *connectors* act as the primary information channels. In this respect, the connectors are used to bind the corresponding DMCs to the operators. In other words, the connectors define the data-flow within context plug-ins. Furthermore, connectors are also used to

perform *mediation* tasks from the data stored in one DMC to the data required in another one, i.e., extracting particular scopes or dimensions from a data structure and/or performing IRO transformations (see subsection 4.4.3). The corresponding mediation is specified through an associated string.

With regard to the data flow, it is assumed that the operators, in conjunction with the connectors, form a *Directed Acyclic Graph* (DAG). This allows the consideration of operators and connectors as a kind of filter chain, which is translated into a single *compute* method. This method deals with all the computation and mediation tasks, without having to cope with loops or other ambiguous effects that can only be hardly grasped at a purely conceptual layer. Further details are presented in the two case-study examples presented in chapter 7.

6.3 Operators

This section presents examples of generic and reusable operators. Naturally, developers can either specify and provide custom types of operators to be used in the MDD framework, or they can simply reuse existing ones. In order to provide a better understanding of the generic operators and the functionalities they might provide, three operator examples are presented.

6.3.1 Value predictor

The basic functionality of the *Value predictor* operator is to accumulate numerical values (distributed over a time period) and try to predict their future values. The rationale for having such an operator is to try and predict trends in resources that might affect the operation of an application. For instance, by monitoring the signal strength of a wireless (e.g., WiFi) connection, it is possible to predict situations where the declining signal strength could result in a broken network link (thus enabling you to preemptively perform tasks such as server synchronization).

Based on the nature of the input data, specialized operators can be used to implement any of the following mathematical techniques: *linear extrapolation*, *polynomial extrapolation*, and *French-curve extrapolation*. Currently, prototypes of the *linear* and the *polynomial* techniques have been implemented and packaged as ready-for-use operators.

6.3.2 Image comparator

With the purpose of allowing motion detection via a web-camera, an operator is provided which compares consecutive images and computes their *delta*[1] . A simple implementation of such an operator is straight-forward: The images are compared pixel-by-pixel and their difference is set as the normalized sum of pixel differences. The difference between pixels is easily computed by checking their difference in each of the three color vectors in the case of *Red-Green-Blue* (RGB) encoding.

Using this operator, a motion sensor can be easily constructed using a queue to accumulate the two most recently captured images to be compared, and the *image comparator operator* to process them. The latter compares the two most recent images and generates an event encoding their computed difference. This operator can be further customized by allowing a *threshold* property (*i.e.*, so that an event is generated only if the detected delta exceeds a predefined value).

6.3.3 Kalman filter

A *Kalman filter* [81] provides an efficient method to estimate the state of stochastic systems and can be considered as a special case of *Recursive Bayesian Filtering*. Such filters are able to deal with noisy, missing and partly redundant measurements while minimizing the expected mean squared error. Kalman filters are often used to estimate the position and velocity of objects and thus provide the basis for many tracking systems. In the realm of context-aware systems,

[1] In this case, *delta* is defined as a number in the range [0, 1] which is used to indicate the difference between the compared images

for example, a Kalman filter can be used to compute the position of users from inaccurate GPS readings. Additionally, it can be used to merge measurements that were taken from different sensors with varying accuracy.

The implemented *Kalman filter* operator realizes a linear Kalman filter. For this purpose, it expects meta-data that express the covariance of the provided measurement dimensions. The operator is configurable to allow the estimation of the state of both static and dynamic objects, assuming a constant state change over time.

6.4 UML Profile

The conceptual model presented earlier in section 6.2 defines the main entities to be covered by appropriate modeling elements. However, before presenting their definition, the selection of an appropriate modeling language is first explained. The main consideration was the choice between *General Purpose Modeling Languages* (such as UML or XML) and *Domain Specific Languages* (DSL). As UML already provides modeling support for defining the internal structure of components—as parts that are connected through ports in composite structure diagrams—it was decided to reuse those concepts to the greatest extend possible. At the same time, the semantics of general UML are tailored to this approach by defining a UML Profile. Thus, the main task of the UML Profile is to map the conceptual entities to UML modeling elements and to define appropriate stereotypes along with tagged values.

Figure 27 shows the UML Profile corresponding to the conceptual model described earlier in section 6.2. For all the main conceptual entities, appropriate stereotypes are defined and their attributes are included as tagged values. In order to avoid repetition, these paragraphs just highlight the extension of UML meta-classes, to provide a clear understanding of how the conceptual entities are reflected by UML modeling elements.

The stereotype `mContextPlugin` extends `UML Class`. This qualifies it to be used in a composite structure diagram for the definition of its internal mechanisms. Actually it would be sufficient if the meta-class `UML Encapsulated Classifier` would be extended, but this meta-class is not available for extension in many UML modeling tools. In the same way, the stereotype `mContextOperator` extends `UML Class` as well. Here too, `UML Part` would be sufficient for use in UML composite structures diagrams.

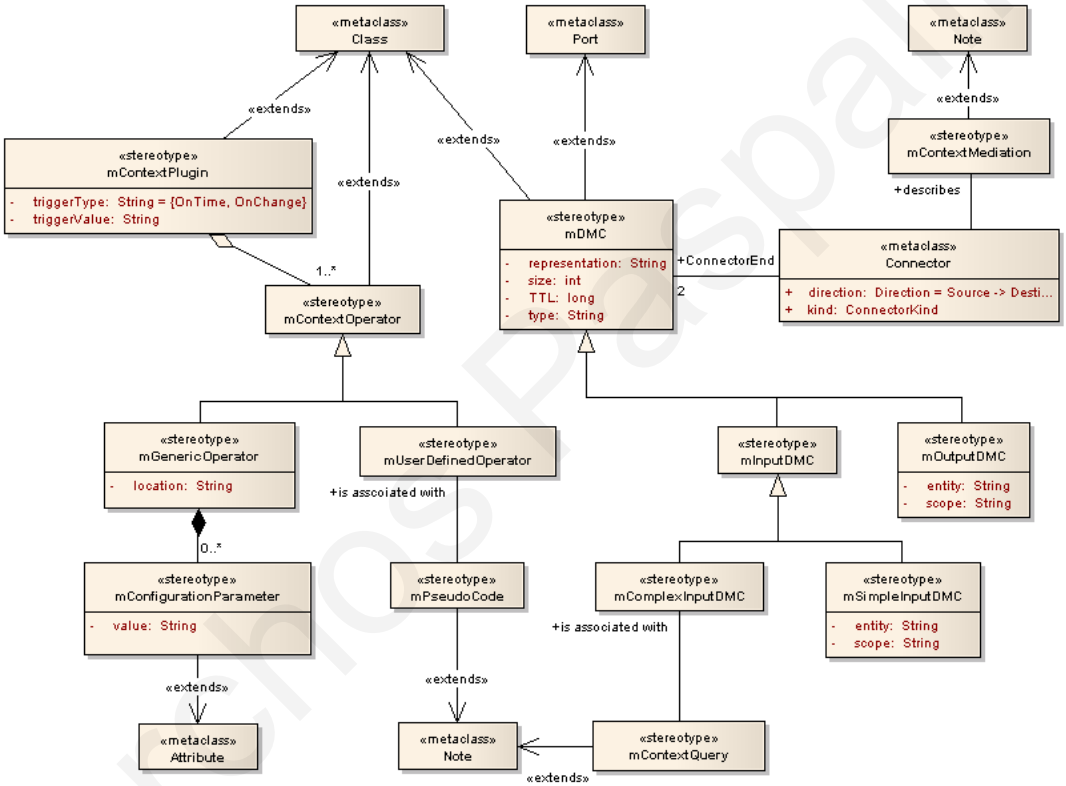

Figure 27: UML profile for modeling context plug-ins.

The stereotype `mConfigurationParameter` extends the meta-class `UML Attribute`. This allows the modeling of the configuration parameters as attributes of the `Operator` class. For user-defined operators the `mPseudoCode` stereotype was introduced, which extends `UML Note`. Thus, the pseudo-code for an operator can be provided by simply associating a note to

the operator class. As DMCs serve as interaction points between the operators, these are modeled as ports. Consequently, a DMC extends the `UML Port`. However, as it is desired to provide a convenient method for modeling the tagged values, the stereotype also extends `UML Class`. For the connector, the standard `UML connector` was used. The attribute `requiredMediation` is reflected through the stereotype `mContextMediation` which extends `UML Note`. This allows the association of the required mediation to the connectors as notes.

6.5 Tool chain and transformation

The main constituents of the proposed tool chain are the *Enterprise Architect* [12] modeling tool and MOFScript [9]. The first is a commercial tool allowing the design of UML-based models. MOFScript is a submission for the MOF-to-text transformation standard and is able to perform the transformation from MOF-based models to arbitrary text formats (which clearly include source code). MOFScript was developed as part of the MODELWARE project [8] and is available as an Eclipse plug-in. Enterprise Architect supports OMG UML 2.x, but is not exactly compliant to the Ecore interpretation of UML2, which is required as the input format of MOFScript. For this reason, an *XSLT Stylesheet* was developed to convert Enterprise Architect exports to models compliant to the Ecore UML2 model.

Developing the MOFScript-based transformation script was mostly straight-forward (see subsection 6.5.1). However, a major challenge arises when in addition to the UML models, further information captured in ontologies, and represented in OWL, must be incorporated into the transformation process. For example, for the mediation tasks performed by connectors it is necessary to have information about the internal structure of the context information and its available IROs. For this purpose, the Ecore meta-model for OWL 1.1 [13] was used, which enables MOFScript to process OWL ontologies. However, a prerequisite is that an appropriate OWL modeling tool

supporting this standard was used. In this case, an OWL modeling tool was used to serve as a standard implementation for the Ecore meta-model mentioned above. In the future, it is planned to provide an appropriate XSLT Stylesheet that performs the transformation from, for example, Protégé exports directly to the Ecore OWL format.

6.5.1 Transformation to Java code

This subsection describes the practical steps that need to be taken in order to produce a context plug-in as a ready-to-be-used component. First, in order to generate the source code for the context plug-in, the UML model presented in section 6.4 needs to be exported in a standard representation. Thus, the model is first exported to XMI and then converted to the XMI/UML2 representation, which is expected as input by MOFScript. Next, the developer imports the resulting model into an Eclipse project. From there, the MOFScript with the provided transformation script is executed, which results to the generation of the actual source code. Besides the source code, the transformation process also creates a directory structure which includes the bundle manifest for the plug-in (required by OSGi), along with its service descriptor declaration (required by the *Declarative Services* system). Finally, a build script is also generated so that it can be directly used by a build tool (such as Apache ANT) to automatically generate the final JAR-packaged bundle.

The transformation script logic is divided in two cycles: First, all the elements available in the model are read and stored in appropriate containers, like lists, hash-tables, etc. This is done to ease the subsequent code generation task which follows immediately after.

Although the core development of the transformation script was straight-forward, two problems have required significant elaboration: First, the correct transformation of the *Directly Acyclic Graph* (DAG) for the generation of the `compute` method of the context plug-in. Second, the automatic generation of source code to support the mediation between connected DMCs.

The first problem was solved by implementing a small recursive algorithm within the transformation script, thus avoiding any pre-processing step. Thereby the main challenge was to implement this algorithm with the limited language support of MOFScript. The algorithm starts with the InputDMCs of the context plug-in. It searches for all connectors with one of these DMCs as a starting point and generates the source-code for these. Afterwards, the algorithm checks all operators whose InputDMCs become connected by the generated connectors, to determine if there exist other connectors with one of these operators' InputDMCs as their target. If such a connector exists, the source code for it is generated. Then, it is possible to include the `compute` method of the connected operators. The algorithm then proceeds recursively with the OutputDMCs of the operator as the starting point. The algorithm terminates when it arrives at an OutputDMC of the context plug-in.

Concerning the second problem, a complete solution is not available yet. In the previous section it was mentioned that an ontology corresponding to the Ecore OWL format can be parsed by MOFScript. However, incorporating the concepts and relationships defined in the ontology into the automatic generation of mediation-methods has not been implemented yet. However, the current version of the transformation tool allows the generation of skeletons for every mediation-method, which is marked with "TODO" mark-ups and appropriate comments in the code, to indicate the points where the corresponding IRO code has to be manually added by the developer.

6.6 Discussion

The main contribution of the proposed model-driven development approach is that it offers the ability to develop sophisticated context sensing plug-ins in a convenient, intuitive and automated way, while at the same time facilitating software reuse. The applicability and the validity of the claimed advantages were justified in two ways. First, it was compared with related work,

as presented in the following subsection. Second, the approach was applied in the context of an undergraduate course at the University of Cyprus. The experiences and recommendations of the participants were then collected in a survey. The results of this experimental evaluation are discussed in chapter 7.

6.6.1 Related Work

A similar model-driven approach for developing context-aware applications is presented by Ayed et al in [23]. In that approach, the authors provide a detailed description of the steps required to generate the code of context-aware applications. These steps cover all the production phases. The authors conclude that the use of MDD in the development of context-aware applications allows platform independent development which "hides the complexity and the heterogeneity of the context-aware and adaptive mechanisms". In line with this work, a similar approach is described in [58] where a *Model-driven Architecture* (MDA) approach is proposed for the development of context-aware applications. This approach proposes six packages of abstractions for facilitating the development of the *component*, the *process*, the *ULCOM* (for ontology definition), the *GUI*, the *data* and the *integration* meta-models. Furthermore, the authors of [134] use MDA-based approaches to design and deploy software applications targeting autonomous robots. MDA is used to produce a PIM of the software, which is then used in combination with a *Platform Model* (PM) to generate the required PSM.

Unlike the proposed MDD approach, the work by Ayed et al [23] aims to enable the design of complete context-aware applications, and thus it puts emphasis on their *variability* aspects. Similarly, the work by Georgalas et al [58] also aims at the design of complete context-aware applications, including their GUIs and business logic. In contrast, the approach presented here focuses on the specification of the actual context plug-ins (i.e., the context sensing and context

reasoning components) as well as on their internal mechanisms which are used to collect and process context information. It is argued that this degree of specialization provides significant advantages as it enables the developers to maximize the portion of the functionality specified in the PIM and as a result minimize the amount of code required in the individual PSMs. Also, the approach presented here is highly geared towards reusability, facilitating the reuse of sub-components (such as operators, DMCs, etc). Regarding the variability of the applications, it is assumed that this is treated by other middleware components as a separate aspect [116]. It should also be noted that the presented MDD approach complements a more extensive MDD-based development framework which facilitates the development of complete context-aware, self-adaptive applications [55].

Finally, it is argued that the use of context plug-ins enables the developers to focus on the concern of context-awareness independently of the other tasks (*e.g.*, GUI and application logic design), thus lowering the complexity and improving their productivity. Similar to our approach, the use of a separate PM in [134] allows the developer to reuse model transformations over several platforms. This is also shown in [53] where the authors use knowledge that is contained in ontologies to automate model transformations used in the model driven development of adaptive services and applications.

6.7 Conclusions and future work

This chapter presented an MDD-based approach for developing context plug-ins. As part of a more comprehensive development suite, this approach provides a number of advantages: First, the MDD-based approach benefits the developers of context-aware applications by improving their productivity. This is a result of their ability to reuse and customize existing models, thus minimizing the amount of code they are required to manually produce. Second, the proposed approach

complements a more extensive development suite (the MUSIC development studio [54]), providing additional support for designing and implementing context-aware applications. Finally, by using MDD, the developers inherit its benefits, i.e. they maximize the amount of functionality that is encoded in the PIM and thus minimize the amount of code that needs to be hand-written. This has the obvious advantage of better maintainability and easier porting across platforms. These advantages were empirically validated by the author and other users of the middleware and the MDD-based approach. They were also experimentally evaluated as part of a set of lab assignments where students were asked to use and compare it to manual (i.e., ad-hoc) implementation.

For the future, it is planned to extend and improve this methodology by implementing additional operators and DMCs and, also, by realizing its support for IROs. Furthermore, this approach will be more extensively evaluated in the context of additional applications. Finally, the custom-tailored component repositories integrated in the modeling tool will be further studied. Such repositories could include standard, reusable operators and DMCs, and could allow the developers to directly select them from inside the development tool (i.e., in a manner similar to how widgets are used for the design of GUIs). Naturally, such an addition would further improve the developer productivity.

Chapter 7

Evaluation

This chapter describes the approach that was followed to evaluate the proposed development methodology and the middleware architecture supporting it. In this regard, two case study applications are presented, while also describing the process followed to develop them as well as how the middleware is used by them. These case-study applications provide the foundations for the evaluation which follows. The evaluation itself consists of two basic components: quantitative and qualitative analysis. The quantitative analysis is performed by examining how the methodology and the middleware are used by developers implementing context-aware applications. This step is realized experimentally in two phases: the developers first use the development methodology and the middleware architecture and then fill-in questionnaires to document their experience. The qualitative analysis is achieved through the analysis of the requirements identified in chapter 3, in relation to the proposed methodology and middleware architecture.

7.1 Developing the Context-aware Media Player case study

The *Context-aware Media Player* (CaMP) case study application was developed to illustrate the use of the development methodology, tools and middleware presented in this thesis. The application consists of a typical media player application which, however, is aware of when the

user is entering—or exiting—their office, so that it can automatically resume—or suspend—the media playback accordingly.

7.1.1 Analysis

From a technical perspective, CaMP is a simple application which only needs to be asynchronously notified when the user enters or exits the room (where CaMP is deployed). In this respect, separating the concern of its business logic (i.e., media playback) from that of its context-aware behavior (i.e., automatically resuming or suspending the playback) is quite straightforward.

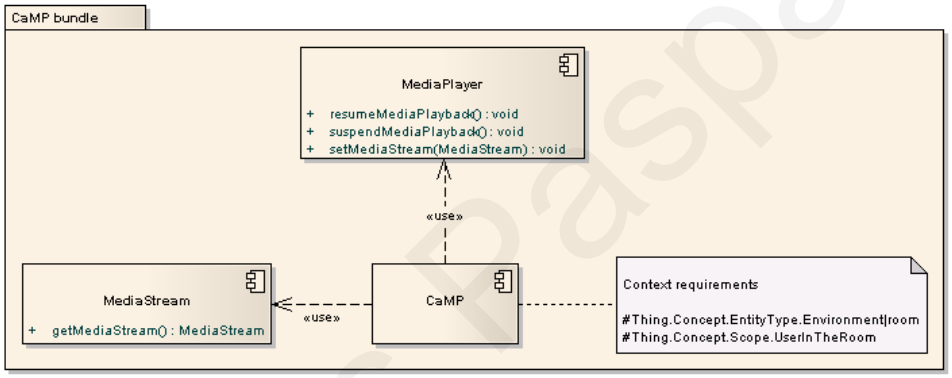

Figure 28: The Context-aware Media Player business logic

The application consists of a central component which makes use of the *Media Stream* and the *Media Player* components. The first is used to get access to a media stream (e.g., a URL of an online music streaming service) and the latter is used to convert the media stream to audible music, played on the device's speakers. It should be noted that the latter also includes methods for resuming and suspending the media playback, which are controlled by the CaMP component. Finally, the CaMP component—and, consequently, the CaMP application—has a dependency on the context type specified by the `#Thing.Concept.EntityType.Environment|room` entity and the `#Thing.Concept.Scope.UserInTheRoom` scope. The three components

illustrated are all packaged in a single OSGi bundle. The architecture of the business logic of CaMP is illustrated in figure 28.

7.1.2 Applying the development methodology

Revisiting the methodology described in section 4.3, the CaMP application can be designed following these steps:

- *Requirements analysis and specification*: This step refers to the analysis of the business logic of the application, a task that was discussed in subsection 7.1.1 and its results are illustrated in figure 28.

- *Analysis and identification of context requirements*: Initially, the only context requirement of CaMP is the context type which describes whether the user is present (i.e., in front of his computer) or not.

- *Selection of appropriate context providers*: As the requested context type is rather complex, the developer decides to implement a custom context reasoner plug-in to provide it.

- *Development of custom context provider*: The corresponding context plug-in for this context type is envisioned as one which takes as input the list of bluetooth devices which are within range (i.e., nearby), along with a measurement of the movement activity sensed in the room and combine them to report when the user enters or exits the room. Its detailed implementation is described in subsection 7.1.3.

- *Repeat analysis and identification of context requirements*: As the selected context plug-in requires additional context types as input, the methodology instructs the developers to revisit this step (R2), in order to handle the new requirements. In this case, the additional context types are the names of the nearby bluetooth devices and the motion sensed in the room. For

the former, an off-the-shelf bluetooth context sensor is reused (see subsection 7.1.4) and for the latter a custom motion context sensor is developed using the MDD approach (see subsection 7.1.5). None of these two context plug-ins have additional context needs, so the developer proceeds with the next step. The resulting architecture is illustrated in figure 29.

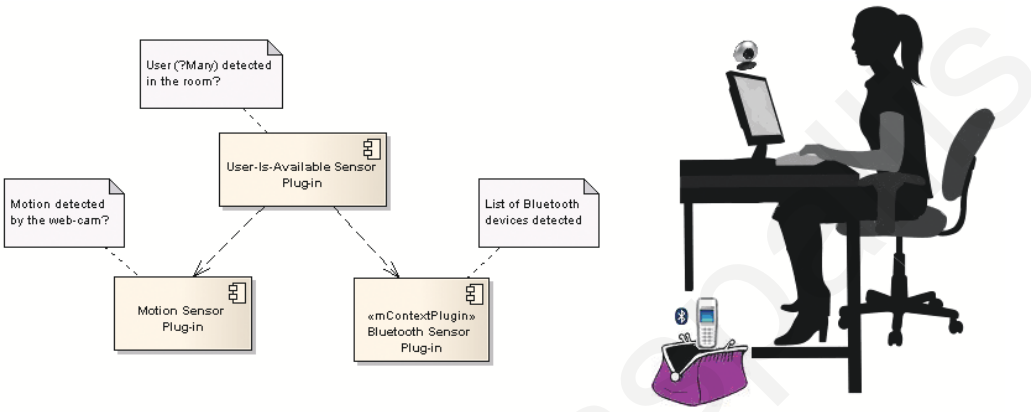

Figure 29: The Context-aware Media Player context-aware logic

- *Binding the business logic to the context providers*: In the case of the CaMP application, this step is rather straightforward. The CaMP bundle is interfaced with the context middleware bundle (see chapter 5). The former uses the context access service to register for asynchronous notification of the requested context type. On the other side, the developed context plug-ins (and the reused one) are simply installed in the context middleware, which takes over their management (i.e., their resolution and activation). The binding of CaMP with the context middleware is more elaborately discussed in subsection 7.1.6.

- *System testing*: This task focuses on testing the context-aware behavior of the application. This step can be facilitated by a context viewer and simulation tool (see section 5.5). These tools enable the developers to view which context types are requested by the installed applications and which ones are provided by the plug-ins. Also, the simulation mode allows

for creating custom context events, while ignoring those created by the actual plug-ins (see last tab in figure 23). This allows to test each context reasoner plug-in individually. For example, in the case of the User-in-the-room plug-in (see figure 29), this can be tested by simulating custom context values for the motion and the bluetooth user types. Finally, a specialized viewer is built in the CaMP application with the purpose of showing how the application's context-awareness logic functions *under-the-hood*. For instance, it will be illustrated in a figure further on that this viewer shows the status of each of the three installed plug-ins at run-time.

- *Production and deployment*: The first of the last two tasks—as specified in the methodology—refers the production and the deployment of the application. In the proposed middleware architecture, the context-aware application (a context consumer) can be deployed either separately or bundled together with the required context plug-ins (the context providers) in the same code package. This decision often depends on the deployment platform and the application itself. For instance, in the case of an application that uses basic context types offered by pre-installed plug-ins (such as the memory, CPU and network sensors), then the application can be packaged and deployed alone. In cases where custom-made context types—or basic context types but of custom high fidelity/precision—are used, then these plug-ins are more suitably packaged and deployed along with the application.

- *Maintenance*: The final step in the development methodology is maintenance. In this task, the developers are expected to monitor their applications (emphasis in their context-aware behavior) and make corrections as needed. For instance, this might involve the fine-tuning of their context-aware behavior by changing the way context is used in them, or by replacing existing plug-ins with others, providing better fidelity or smaller resource footprints. It

should be noted that substitution or updates of context plug-ins can be applied at run-time (as it is also natively supported by the underlying OSGi framework).

From a conceptual point-of-view, the application functions as follows: When the user enters her or his office, their smart-phone is discovered, triggering a context event. At the same time, to make sure that the user is not in a neighboring office but indeed inside their own, a camera-based motion sensor is also used to detect motion in the office. Events from these two sources are combined to generate a higher-level context type, abstracting whether the user is currently inside her or his office or not. These three context types correspond to three context plug-ins, as shown in figure 29.

The following subsections describe how these context types are provided by plug-ins which are either newly developed or reused.

7.1.3 Developing the User-in-the-room context reasoner plug-in

The first plug-in that is developed is the one detecting whether the user is inside the room where CaMP is deployed, or not. As plug-ins are normal OSGi bundles, the first step is to define their manifest file, which is the one specifying the static dependencies of the plug-in (i.e., which libraries are required, beyond the standard ones provided by the JVM).

The manifest file shown in listing 7.1 shows that the plug-in has static dependencies on libraries offered by the middleware (i.e., packages `org.istmusic.mw.context.events` to `org.istmusic.mw.context.ontologies`) and libraries implementing the bluetooth and motion context models offered by the corresponding plug-ins. Furthermore, the manifest specifies that the bundle enclosing this plug-in exports a package which implements the model used internally by the plug-in (i.e., `cy.ac.ucy.cs.osgi.context.user_in_the_room.model`).

As context plug-ins are also normal OSGi components controlled by the *Declarative Services* specification, the next step in their development is the specification of the service descriptor file. This descriptor specifies which services are provided, and which ones are required by the plug-in. All context plug-ins provide one service (i.e., the `IContextPlugin` shown in listing 5.1 and discussed in section 5.2.2).

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: User-in-the-room sensor context plugin
Bundle-SymbolicName: UserInTheRoomSensor
Bundle-Version: 1.0.0
Bundle-ClassPath: .
Import-Package:
org.istmusic.mw.context.plugins,
org.istmusic.mw.context.events,
org.istmusic.mw.context.util.scheduler,
org.istmusic.mw.context.model.api,
org.istmusic.mw.context.model.impl,
org.istmusic.mw.context.model.impl.values,
org.istmusic.mw.context.ontologies,
cy.ac.ucy.cs.osgi.context.bluetooth.model,
cy.ac.ucy.cs.osgi.context.motion.model,
javax.swing
Export-Package:
cy.ac.ucy.cs.osgi.context.user_in_the_room.model
Service-Component: OSGI-INF/UserInTheRoomSensor.xml
```

Listing 7.1: The manifest file of the User-in-the-room plug-in

Furthermore, the service descriptor includes optional parameters which, in this case, are used to specify the context types offered by the plug-in. In the case of context sensors, only provided context types are specified, while in the case of context reasoners—such as the *User-in-the-room* plug-in—the required context types are also specified.

```
<?xml version="1.0"?>

<component name="UserInTheRoom" immediate="true">

  <implementation class="...user_in_the_room.plugin.UserInTheRoomSensor"/>

  <service>
    <provide interface="org.istmusic.mw.context.plugins.IContextPlugin"/>
  </service>

  <property name="PROVIDED_CONTEXT_TYPES" value="UserInTheRoom"/>
  <property name="PROVIDED_CONTEXT_TYPE_ENTITY[UserInTheRoom]"
      value="#Thing.Concept.EntityType.Environment|room"/>
  <property name="PROVIDED_CONTEXT_TYPE_SCOPE[UserInTheRoom]"
      value="#Thing.Concept.Scope.UserInTheRoom"/>

  <property name="REQUIRED_CONTEXT_TYPES" value="motion_bluetooth"/>
  <property name="REQUIRED_CONTEXT_TYPE_ENTITY[motion]"
      value="#Thing.Concept.EntityType.Environment|room"/>
  <property name="REQUIRED_CONTEXT_TYPE_SCOPE[motion]"
      value="#Thing.Concept.Scope.MotionDetected"/>
  <property name="REQUIRED_CONTEXT_TYPE_ENTITY[bluetooth]"
      value="#Thing.Concept.EntityType.Device|this"/>
  <property name="REQUIRED_CONTEXT_TYPE_SCOPE[bluetooth]"
      value="#Thing.Concept.Scope.Bluetooth.AttachedDevices"/>

</component>
```

Listing 7.2: The service descriptor of the *User-in-the-room* plug-in

Listing 7.2 illustrates the service descriptor for the user-in-the-room plug-in. This descriptor shows that a single context type is provided, as discussed in the analysis of the plug-in (see subsection 7.1.1). Furthermore, the same listing shows that two context types are required as input. These are the motion and the attached bluetooth devices.

7.1.3.1 Realizing the context model

Revisiting the basic modeling concepts (see subsection 4.4.2), a custom model for the user-in-the-room plug-in is realized. The model consists of a single context element encapsulating a single context value. This context element (and the context value) abstract the information of

whether the specified user is in the room as a `boolean` value. Furthermore, two special metadata attributes are defined and attached to the context value: *timestamp* and *accuracy*. The first is used for specifying the time at when the context value was created, and the latter for estimating its accuracy (e.g., as a scalar in the range [0, 1]). This model is illustrated in figure 30.

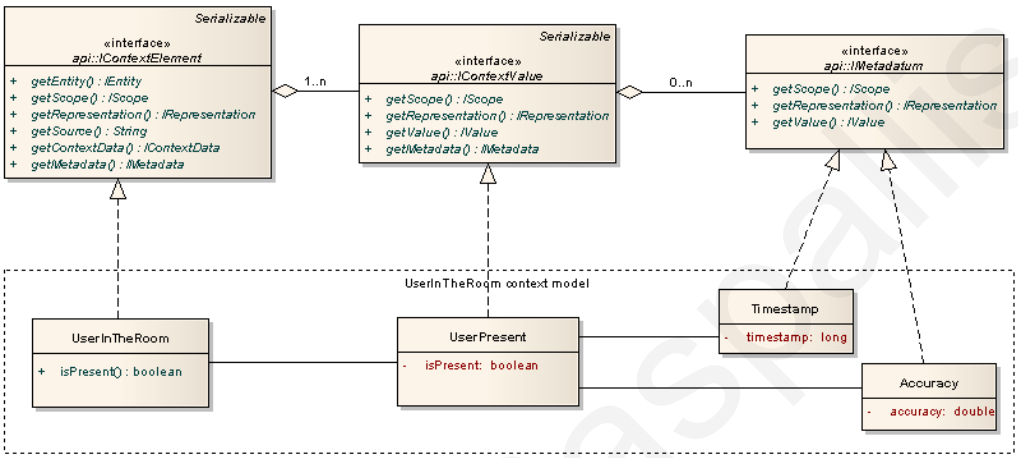

Figure 30: The context model used in the User-in-the-room plug-in

7.1.3.2 Implementing the plug-in

The context middleware provides two abstract classes that can be extended (using standard Object-Oriented inheritance) to realize context sensor and context reasoner plug-ins. Besides realizing the fundamental functionality specified in the `IContextPlugin` interface, the class defined as `AbstractContextPlugin` also provides a predefined method for *firing* context events. When the plug-in's context sensing logic is implemented, it uses this method to communicate context events to the middleware. Plug-in realizations also implement the `activate` and `deactivate` methods, which are automatically controlled by the middleware (see subsection 5.2.4), so that the sensing is resumed when activated and suspended when deactivated.

The `AbstractContextReasonerPlugin` extends the `AbstractContextPlugin` to add functionality so that the plug-in metadata is automatically read (i.e., the provided and required context types, as specified in the service descriptor file, such as the one shown in listing 7.2). Given the plug-in's context dependencies, a mechanism embedded in this class registers the plug-in as a listener for asynchronous notification for each required type specified in the service descriptor. A special operation—the `contextChanged` method—is realized by the inheriting plug-in to be used to collect the context events.

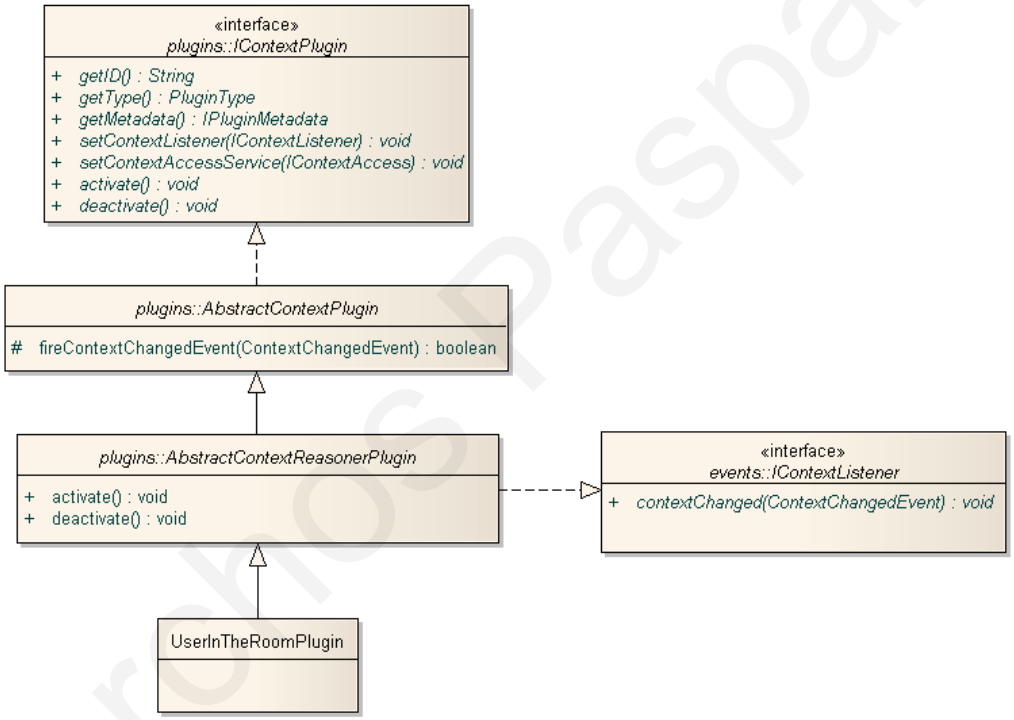

Figure 31: The class hierarchy of the User-in-the-room plug-in

The user-in-the-room plug-in and its position in this hierarchy are illustrated in figure 31. The implementation of a context reasoner plug-in is completed by realizing the logic that collects the incoming context events, and processes them to generate the outgoing ones. In this case, the plug-in logic should check for when the bluetooth device associated with the user is discovered to be

nearby (i.e., within bluetooth range) and when not. However, to eliminate cases where the user is nearby, but not in front of her or his computer, a second value is checked to make sure that motion was detected in front of the computer. This simple logic can be realized as shown in listing 7.3.

```
public class UserInTheRoomSensor extends AbstractContextReasonerPlugin
{
  public static final long MOTION_VALIDITY_PERIOD = 10000L; // 10 seconds
  ...
  private long lastMotionTimestamp = 0L;
  private double accuracy = 0.0d;
  private boolean userBluetoothDeviceAttached = false, userInTheRoom = false;

  public void contextChanged(ContextChangedEvent event)
  {
    ...
    if(contextElement instanceof MotionContextElement) {
      lastMotionTimestamp = System.currentTimeMillis();
      accuracy = (MotionContextElement) contextElement.getAccuracy();
    } else if (contextElement instanceof BluetoothContextElement) {
      BluetoothContextElement bce = (BluetoothContextElement) contextElement;
      userBluetoothDeviceAttached = bce.contains(...getUserDeviceID());
    }

    boolean motionDetectedRecently = lastMotionTimestamp +
        MOTION_VALIDITY_PERIOD > System.currentTimeMillis();
    if(motionDetectedRecently && userBluetoothDeviceAttached) {
      if(!userInTheRoom) {
        userInTheRoom = true;
        UserInTheRoomContextElement ce = new UserInTheRoomContextElement(...,
            userInTheRoom, Math.min(0.8, accuracy * 2));
        fireContextChangedEvent(this, ce);
      }
    } else if(!userBluetoothDeviceAttached) {
      if(userInTheRoom) {
        userInTheRoom = false;
        UserInTheRoomContextElement ce = new UserInTheRoomContextElement(...,
            userInTheRoom, 0.8);
        fireContextChangedEvent(this, ce);
      }
    }
  }
}
```

Listing 7.3: The code implementing the User-in-the-room plug-in

In this implementation, whenever the User-in-the-room plug-in receives a new context event (via the `contextChanged` method), it first checks which kind of event it is and then it fires a newly created event, if needed. In the first part of the algorithm, two parameters are updated: the `lastMotionTimestamp` long value and the `userBluetoothDeviceAttached` boolean value. In the second part of the algorithm, an intermediate value is first computed. The boolean parameter `motionDetectedRecently` abstracts whether a motion event has been sensed *recently* (i.e., within the validity period which is 10 seconds). Finally, if the user's current state (i.e., `userInTheRoom`) is not set as true, then when their bluetooth device is discovered and motion is recently detected, the state changes to true. Otherwise, when the current state is true, but the user's bluetooth device is disconnected, the `userInTheRoom` changes back to false. In both case, an appropriate context change event is created and fired.

A notable detail refers to the construction of the context element. The custom context element was defined to include a single boolean value, but also two metadata parameters (see figure 30). The first metadata parameter, the *timestamp*, can be easily computed using standard library calls. The second parameter, the *accuracy*, requires a more elaborate approach. While the developers are free to assign any value they want (e.g., a constant value of 0.5), in principle they should try to assess the accuracy of the generated context data as a function of the accuracy of the input. For example, assuming that bluetooth adapters have a limited accuracy of 0.8, then the accuracy can be fixed to that value when the user bluetooth device is detached and a `userInTheRoom=false` event is reported. On the other hand, when both the bluetooth and the motion input context data are used as input for the `userInTheRoom=true` event, then the accuracy should be a function of both input context types. In this example, the accuracy is computed as twice the accuracy of the motion sensor with a maximum value of 0.8, which is the bluetooth plug-in's maximum accuracy.

While the development of this plug-in is relatively straightforward, it includes multiple steps which could be rather automated. For instance, the definition of the manifest file and the service descriptor, and even the construction of the context model itself could be all handled automatically by appropriate development tools. This can be achieved with the MDD-based development approach presented in chapter 6), as it illustrated via an example in subsection 7.1.5.

7.1.4 Reusing the Bluetooth context sensor plug-in

The Bluetooth sensor plug-in monitors and reports whether a predefined bluetooth device (e.g., a smart-phone carried by the user) is near her or his office (simple detection by the adapter fixed on the workstation computer is sufficient as typical bluetooth adapters have a range of just a few meters). In this way, this plug-in senses whether the specified smart-phone, and thus its owner, is nearby with relatively high accuracy. In practice, this approach achieves functionality similar to that of an RFID sensor detecting tags, assuming—of course—that the users always carry their smart-phones along and their bluetooth adapters are on.

As it was mentioned in subsection 7.1.2, for this context plug-in, an existing implementation is selected from a public repository. A context plug-in repository is a public directory listing existing, reusable plug-ins that can be used as off-the-shelf components. The attributes of the plug-ins (e.g., supported platforms, limitations, etc) are also listed along with the plug-ins. An example of such a repository was implemented in the scope of the MUSIC project [6].

7.1.4.1 The MUSIC context plug-in repository

The MUSIC context plug-in repository is a manual implementation (see subsection 4.3.1), allowing the developers to browse a list of available plug-ins and freely download and reuse those which are useful for their applications.[1]

[1] http://www.ist-music.eu/MUSIC/developer-zone/context-plug-ins-repository-1

Table 6: The context plug-ins listed in the MUSIC repository

Plug-in name	Description	Attributes
Motion sensor	The motion sensor plug-in uses a web-camera to detect and report any motion sensed in the area viewed by the camera.	Author: Nearchos Paspallis (University of Cyprus)
Noise sensor	The noise sensor plug-in uses a microphone to detect and report the sensed noise.	Authors: Simos Gerasimou, Andreas Christodoulides and Nearchos Paspallis (University of Cyprus)
Bluetooth sensor	The Bluetooth sensor plug-in uses a hardware Bluetooth adapter to detect active, neighboring bluetooth devices.	Author: Nearchos Paspallis (University of Cyprus)
RFID sensor	The RFID sensor plug-in uses a hardware RFID adaptor to detect nearby RFID tags.	Authors: Eduardo Soladana and Jorge Lorenzo (Telefonica I+D)
Connectivity sensor	The Connectivity sensor plug-in checks if there is an active connection to internet.	Authors: Michalis Agathokleous, Christos-Aimilios Pras, Christodoulos Efstathiades and Christos Louka (University of Cyprus)
Location-by-IP sensor	The Location-by-IP sensor plug-in computes your location based on your IP. Typically, this returns a high-level result (like for example the city-country, e.g., "Nicosia-Cyprus").	Authors: Michalis Agathokleous, Christos-Aimilios Pras, Christodoulos Efstathiades and Christos Louka (University of Cyprus)
Calendar sensor	The Calendar sensor plug-in periodically communicates with the Google Calendar service and fetches events from the user's personal calendar.	Authors: Panagiota Hapoupi, Melinos Averkiou and Savvas Michael (University of Cyprus)
WiFi-based location sensor	Checks the ID of nearby WiFi routers (e.g., MAC address) in order to determine the location of the device.	Authors: Rolando Sergio, Cristina Fa and Massimo Valla (Telecom Italia Lab)
GSM-based location sensor	Uses the ID reported by nearby GSM cell towers to detect the location of the device.	Authors: Rolando Sergio, Cristina Fa and Massimo Valla (Telecom Italia Lab)
Resource sensor	A specialized (MUSIC middleware-specific) sensor that wraps the resource plug-ins to provide read-only information on battery, memory and network.	Authors: Paolo Barone and Alessandro Mamelli (HP Italy)

A list of the plug-ins available in this repository (as of August 15th, 2009) is shown in table 6. Currently only web-access of the context plug-in repository is supported. However, a more intelligent, run-time use of the repository is also a valuable property. For instance, existing component repositories could be reused to enable a dynamic mechanism which searches, analyzes and downloads appropriate plug-ins at run-time. A suitable repository could be, for example, the OSCAR OSGi component repository.[2]

For the purposes of the CaMP application, the Bluetooth sensor plug-in is downloaded and reused by installing it to the OSGi run-time in its default, packaged form.

7.1.5 Developing the Motion sensor plug-in

The Motion sensor plug-in is a context sensor that periodically takes pictures from a camera and estimates the motion sensed in the environment by computing their difference. In order to demonstrate the approach presented in chapter 6, the motion sensor is implemented using the model-driven development approach.

7.1.5.1 Using the UML profile

In the MDD approach, the context plug-ins are modeled using the UML Profile (see section 6.4). This approach is also viewed in comparison to the manual approach for implementing a context plug-in, as it was presented in subsection 7.1.3 with the User-in-the-room plug-in.

First, the context plug-in itself and its associated `Input DMC`s and `Output DMC` are modeled appropriately. This is done in a *UML Class Diagram* as shown in figure 32. The Motion sensor plug-in is modeled through a class with the stereotype `mContextPlugin`. The plug-in is parameterized by defining the trigger type: a change event in the `Input DMC` (i.e., a new element is inserted in the `Input DMC`). In practice, the `MotionDetectorPlugin` is associated with

[2] http://oscar-osgi.sourceforge.net

a `SimpleInputDMC` named *Input* and one `OutputDMC` named *Output*. Both DMCs are modeled as classes with the corresponding stereotypes. The `Input DMC` is parameterized with just a simple variable that specifies unlimited `TTL` for its elements (i.e., the stored element will never be automatically invalidated). Additionally, it is specified that the *Input* will request context information characterizing the entity `Environment|room`. The scope of the requested information is `ImageFromWebcam` and the requested representation is `ImageFromWebcamDefaultRep`. On the other hand, the `Output DMC` is configured to specify that the information provided by the context plug-in to the middleware also refers to the entity `Environment|room`. Its scope is defined as `MotionDetected` and it is encoded using the `MotionDetectedDefaultRep` representation.

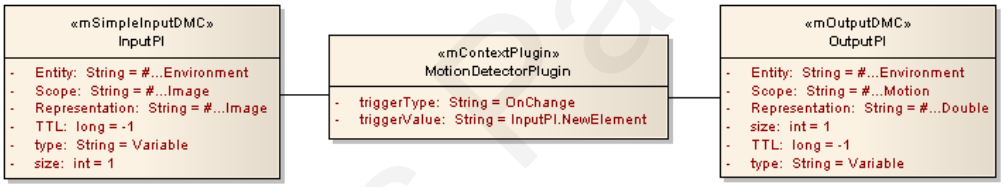

Figure 32: UML class diagram of the Motion sensor plug-in

After the context plug-in has been defined as described above, the developers need to model the operators which are needed inside the plug-in. This step is also done using a UML class diagram. In the case of the Motion sensor plug-in, a single operator is required: the generic `ImageComparingOperator`. Figure 33 shows the corresponding UML class diagram including its definition and parameterization.

The model of the operator is similar to the one of the context plug-in itself. Their main differences are that, in contrast to the context plug-in, an operator cannot be triggered independently and, thus, the `ImageComparingOperator` does not specify any `triggerType` or

triggerValue. Instead, as a generic operator, the ImageComparingOperator provides the package location of the corresponding class to be instantiated. Another difference compared to context plug-ins is that an operator only works on generic DMCs that do not directly interact with the context middleware. Hence, the DMCs only specify the representation (i.e., the data type) of the contained elements. It should be finally noted that as the ImageComparingOperator works on two *consecutive* and *fresh* images, the Input DMC is used to cache two images only. Thus, the DMC is defined as a queue of size 2 and a TTL of 1000ms.

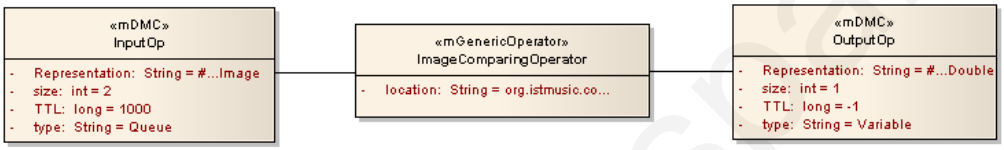

Figure 33: UML class diagram of the Image comparing operator

After the context plug-in and the operator have been specified, the next step is to model the data-flow between the plug-in and the operator. This is done by connecting the defined DMCs and by specifying the necessary mediation tasks. For this purpose, a *UML Composite Structure Diagram* is used, as illustrated in figure 34.

In this diagram, the previously defined ImageComparingOperator is modeled as a nested classifier (represented as UML Class) of the Motion Sensor class and the corresponding DMCs are connected through directed *UML Connectors*. In addition to the pure data flow, *UML Notes* associated to the connectors specify the needed mediation tasks. The note associated to the *Input-to-Input Connector* has the meaning that from the context element stored in the Input DMC of the Motion sensor, the scope ImageBuffer in the representation BufferedImage has to be extracted and inserted into the Input DMC of the ImageComparingOperator in

the same representation (i.e., `BufferedImage`). A mediation is also specified for the Output-to-Output connector.

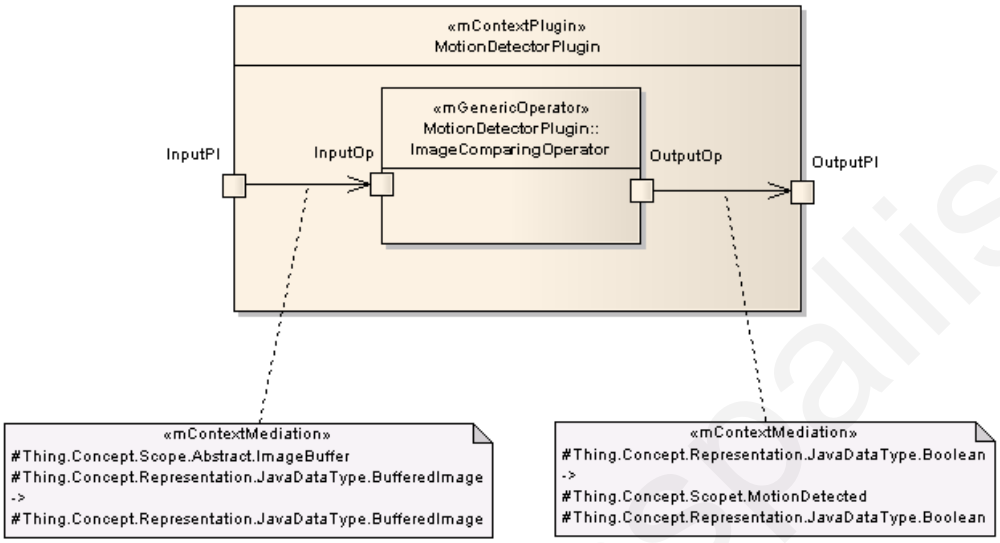

Figure 34: UML composite structure diagram of the Motion sensor plug-in

7.1.5.2 Transformation to Java code and packaging

Having defined the UML-based model, the next step is to transform it into Java code. For this purpose, the developers follow the steps described in section 6.5.1. While this step is mostly automated, the IRO transformation parts of the resulting code are incomplete, requiring that the developers specify them manually.

Once the transformation step is completed, the developers have the source code of the plug-in available. As mentioned above, minor coding might still be needed (in case the context data need inter-representation transformation, in which case the modeling API can be used). Once the final pieces of code are added, the plug-in is compiled and packaged in a JAR-based bundle so that it is ready for deployment. As mentioned before, the manifest, service descriptor, source code, and

even a custom build file are all generated automatically so that the developer needs not spend any time beyond the actual plug-in design.

7.1.6 Binding with the middleware

Having acquired the three plug-ins to be used for providing the context information required by the CaMP, the next step is the binding between the CaMP and the context middleware. This binding ensures that the intended context-aware behavior of the application is correctly implemented.

```
<?xml version="1.0"?>

<component name="CaMP">

  <implementation class="cy.ac.ucy.cs.camp.CaMP"/>

  <reference name="context_access"
    interface="org.istmusic.mw.context.IContextAccess"
    bind="setContextAccess"
    unbind="unsetContextAccess"
    cardinality="0..1"
    policy="dynamic"/>

</component>
```

Listing 7.4: The service descriptor of the Context-aware Media Player

Just like any other OSGi bundle, CaMP defines a manifest and a service descriptor. The former is a typical manifest file (omitted for brevity) and the latter is a simple expression of CaMP's dependency on the context access service. The XML code of this service descriptor is shown in listing 7.4. It should be noted that this listing specifies the context access service as *optional* (i.e., its cardinality is 0..1), which means that the application can be deployed and launched even in the absence of the context middleware. This is possible, because in the case of the CaMP application the context-aware behavior is treated as *additional* rather than *required* functionality.

Finally, the code implementing the context-aware behavior of CaMP is shown in listing 7.5 (some details were omitted or simplified to avoid cluttering). Note that the only link to the requested context information is established via the specified *entity* and *scope* parameters (the default representation is implied).

```
public class CaMP implements IContextListener
{
  public static final IEntity ENTITY
    = Factory.createEntity("#Thing.Concept.EntityType.Environment|room");
  public static final IScope SCOPE
    = Factory.createScope("#Thing.Concept.Scope.UserInTheRoom");

  private final MediaPlayer mediaPlayer = new MediaPlayer();

  public CaMP() { ... // init the GUI }

  public void setContextAccess(IContextAccess contextAccess)
  {
    contextAccess.addContextListener(ENTITY, SCOPE, this);
  }

  public void unsetContextAccess(IContextAccess contextAccess)
  {
    contextAccess.removeContextListener(ENTITY, SCOPE, this);
  }

  public void contextChanged(ContextChangedEvent event)
  {
    IContextElement contextElement = event...;
    IValue value = contextElement.getContextData().getContextValue(SCOPE).
        getValue();
    BooleanValue booleanValue = (BooleanValue) value;
    if(booleanValue.getBooleanValue().booleanValue())
      mediaPlayer.start();
    else
      mediaPlayer.stop();
  }
}
```

Listing 7.5: The implementation code of the Context-aware Media Player

When connected to—or disconnected from—the context middleware, the `setContextAccess` method—or the `unsetContextAccess` method respectively—is automatically invoked by the *Declarative Services* run-time (i.e., see the `bind` abd `unbind` clauses in listing 7.4). When the first method is invoked, the application uses the context access service reference to register itself for asynchronous notification of the specified entity/scope pair. When the second method is invoked, the application uses the same service reference to cancel the previous registration. This behavior allows the context middleware to be aware of the context needs of the application and thus automatically handle the lifecycle of the corresponding plug-ins.

When context events of the requested type are created, those are communicated through the middleware to the CaMP application via the `contextChanged` method. In this case, this method is simply used to extract the boolean value abstracting whether the user is in the room or not. Based on the value encoded in the received event, the media player is started (i.e., resumed) or stopped (i.e., suspended) accordingly.

Finally, as shown in figure 35, the CaMP application also provides manual control for resuming and suspending the media playback. In this respect, the context-aware behavior of the application can be thought of as an additional feature, allowing for the automatic control of the media player on behalf of the user, as per the ubiquitous computing paradigm, but without completely taking the control out of the user's hands.

7.1.7 Deploying the Context-aware Media Player

The three context plug-ins and the CaMP application are all packaged as individual JAR-based OSGi bundles which are installed along with the context middleware (which is packaged as an OSGi bundle itself).

Once the plug-ins are installed, they are automatically discovered by the context middleware which registers their metadata and attempts to resolve them (see subsection 5.2.3). When the CaMP application is started, the context access service is bound with it and the former subscribes for notification of relevant context events. This triggers the context middleware to reevaluate the offered and needed context, and activate the three plug-ins as needed (see subsection 5.2.4).

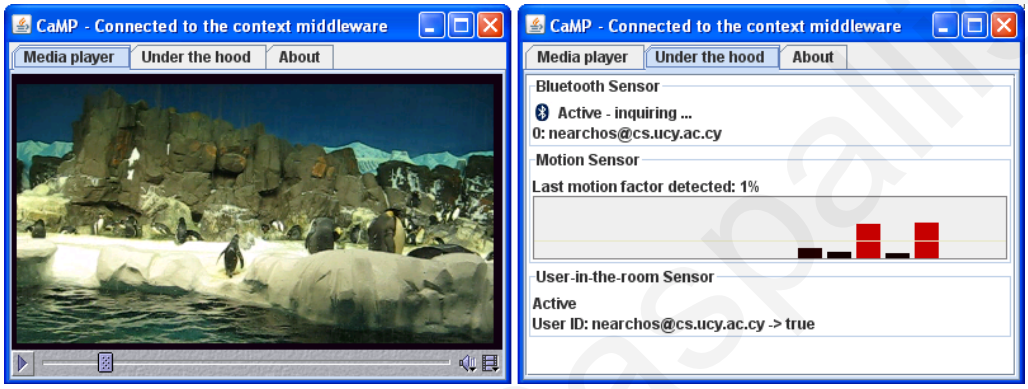

Figure 35: Screenshots of the Context-aware Media Player

The appearance of the application's tabs is shown in figure 35. The first tab shows the media player and its manual controls (a single button which resumes or suspends the playback). The latter provides an *under-the-hood* view of the plug-ins for demonstration purposes. At the top is the Bluetooth sensor viewer showing that the plug-in is active and that it has discovered a Bluetooth device identified as "nearchos@cs.ucy.ac.cy". The second viewer shows that the Motion sensor plug-in is also active and further displays a histogram with the recent motion values reported (two of them exceeding the threshold). Finally, the User-in-the-room sensor viewer at the bottom simply displays a boolean value indicating if the predefined user (in this case the author of the thesis) is detected to be present or not. As the application is running, the user can enter or exit their room, with the application adapting accordingly based on the changing context detected by the plug-ins.

7.2 Developing the Signal Strength Predictor case study

The next case-study refers to the development of a *Signal Strength Predictor* (SSP) system. The system is based on two plug-ins which can be combined to predict the user location first and given that information, make a prediction for the upcoming wireless signal strength. In both cases, the context repository is utilized to consider historical context values in order to make a prediction.

This case-study consists of the `LocationPrecictorPlugin`, which is a sensor plug-in, and the `SignalStrengthPredictorPlugin`, which is a reasoner plug-in. The former aims at predicting the future location of the device or the user. In this case, this is done by querying the context history (i.e., the context repository) for past locations and then applying linear (or polynomial) regression analysis on it. In the simple example presented in the following paragraphs, a simple linear regression approach is selected (it is assumed that the user moves on a nearly straight line). In real-world scenarios, this plug-in could be significantly more sophisticated, utilizing additional information such as the user's calendar, map-data, etc. The latter is also a predictor plug-in, but is based completely on the context history. Given a prediction of the future location, this plug-in aims to predict the signal strength of the wireless network. It does so simply by checking historical values of the network signal strength while the user was visiting the predicted location (in a previous occasion). Given these values, the plug-in computes the average of the historical signal strength values (recorded at the corresponding location) and triggers and event.

Arguably, the resulting context information type can be used in a number of different case-study applications. For instance, a variant of the CaMP application, presented earlier in section 7.1, is designed to play media streamed by a server. For instance, instead of opening a local media file, CaMP connects to a media server—such as a web-streaming radio—and receives data which is temporarily stored in a buffer and then played. This application utilizes the SSP information to

adjust the buffer size of the network-based streaming component. When the user moves towards a low-quality network, the buffer size is increased to better accommodate the network anticipated instability. On the other hand, when the user moves in a well-connected area, the buffer size is reduced to free useful memory resources for use by other applications.

The following subsections describes the development of the two context plug-ins needed in the scenario. Rather than displaying the packaging or modeling details of these plug-ins—such as their service descriptor and manifest files or the custom context elements developed—the following section concentrates on the context reasoning mechanisms and in particular on the context queries.

7.2.1 Developing the Location predictor plug-in

The location predictor plug-in uses a simple linear regression method to predict the location in the next 10 seconds. To achieve this, the plug-in periodically queries the context repository for all the stored locations in the last 100 seconds. If a sufficient number of values is found (e.g., over 5), then a prediction is made for the following 10 seconds.

In order to be possible to use the linear regression method, a linear change of location is assumed (i.e., the user walks on a straight line). While this assumption is not quite realistic, it is nevertheless sufficient for demonstrating the purpose of the context repository and the hierarchical architecture of the context plug-ins. In practice, more complex methods could be used for predicting the location, including the utilization of information from the user's calendar, or the application of a more sophisticated, polynomial extrapolation method.

In linear regression, it is assumed that a number of (timed) sample points are available, which are considered to be roughly laid on a straight line. Given a two-dimensional space, it is desired to devise a function that given the coordinate x of a point near the sample points, it can compute its coordinate y. Assuming that a total of N sample points are available, then the function into

question can be expressed as $f : y = \beta_0 + \beta_1 x$, where $\beta_1 = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$ and $\beta_0 = \bar{y} - \beta_1 \bar{x}$ (in this equation the $\bar{x}$ and $\bar{y}$ are the average values of x_i and y_i respectively). Given these equations, the corresponding functions are easily implemented in Java as plain methods. The same method (and equations) are applied twice: once for estimating the X and another time for the Y coordinate. In both cases the x axis corresponds to the time and the y axis to the X and Y coordinate respectively.

To access the requested context information, the context plug-in either registers for asynchronous notification of the location context type or, alternatively, it issues a conditional context query which encodes the request.

```
<?xml version="1.0"?>

<ctxQuery resultName="locationHistory">

  <entity ontConcept="#Thing.Concept.EntityType.Device">this</entity>

  <scope ontConcept="#Thing.Concept.Scope.Location"
      ontRep="#Thing.Concept.Representation.Coordinates">
    location
  </scope>

  <action type="SELECT">
    <conds>
      <cond type="ONVALUE">
        <constraint par="location.#Timestamp" op="GT" value="2496740706421"/>
      </cond>
    </conds>
  </action>

</ctxQuery>
```

Listing 7.6: The context query used by the Location predictor plug-in

An example of such a request expressed in XML is illustrated in listing 7.6. This query specifies the requested entity, scope and representation which correspond to the user location expressed as coordinates. The actual selection in the query is expressed with the `SELECT` action

which includes a single condition. This condition checks that the value identified by the name `Timestamp` (a metadata value) is greater (encoded as `GT`) than a predefined threshold. In this case, the threshold varies as different queries are placed at different times, and every time the threshold is set as 100 seconds before the current time.

Once the β_0 and β_1 parameters are computed, the plug-in applies an extrapolation and gets an estimate for the X and Y coordinates for the timestamp 10 seconds in the future. Assuming that the predicted location is not too close to the current one (i.e., the user is moving), an event is fired notifying the middleware of the predicted value. This value is identified with a special context scope (i.e., `#Thing.Concept.Scope.LocationPrediction`). This clarifies that the encoded information is a *prediction* for a future value, allowing context consumers to understand its semantics and exploit it.

Because the location predicted by such a plug-in is often inaccurate, the plug-in should also assign metadata to its predicted values, characterizing their *accuracy*. Such metadata values have their own scope and representation, and are directly associated to either the context element or the corresponding context value.

Finally, while several methods can be employed for the computation of the actual accuracy value, a simple solution is realized by computing the *standard deviation* of the sample points used in the prediction from the computed line. In this way, when the sample points are found to be forming a straight line (e.g., the user moves in a straight aisle), the accuracy is assumed to be high.[3]

[3]Standard deviation is a metric used in probability theory and statistics to indicate whether the data points in a sample set are close to a median value or not.

7.2.1.1 Alternative implementation

An alternative implementation of the Location predictor plug-in is also possible via the use of the MDD approach described in chapter 6. With this approach, the plug-in is modeled in UML, as shown in figure 36, and the model is used to automatically generate the plug-in through code transformation (see subsection 6.5.1).

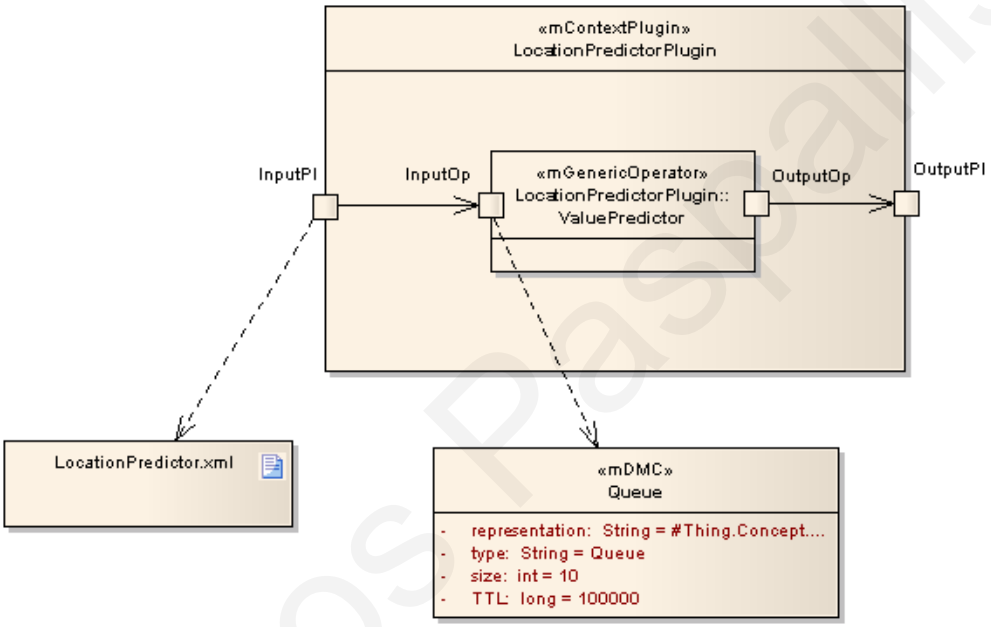

Figure 36: The UML model of the Location predictor plug-in

Instead of describing the detailed steps required for the development of the plug-in, this subsection rather concentrates on some noteworthy details (a more extensive description of this process was illustrated earlier with the development of the Motion sensor plug-in in subsection 7.1.5).

First, for the development of the Location predictor plug-in, the generic Value predictor operator (see subsection 6.3.1) is used. This operator implements the linear regression logic described earlier, but it is packaged according to the standard operator interface that makes its functionality easily reusable.

Second, the input DMC is associated with a context query expressed in XML. The query is practically equivalent to the one presented in listing 7.6. Based on this input, the transformation tool generates code which periodically accesses the context repository as per the XML-based query, and uses the result to feed the operator's input port (a limited-size queue).[4]

Third and final point is that no mediation is required between the plug-in's input port and the operator's input port. The reason for this is that the correct input representation is already defined in the query, which triggers the context system to perform the mediation itself, and convert the location data to coordinates-based, if not in this form already.

7.2.2 Developing the Signal strength predictor plug-in

Having defined the location predictor sensor plug-in, the next step is to define how that information is utilized to also predict the upcoming signal strength. As noted already, the logic of this plug-in is simply based on consulting stored context information and associating the predicted location with a known (i.e., measured) signal strength.

Unlike the Location predictor, which is a *sensor* plug-in, the Signal strength predictor is realized as a context *reasoner*. As such, it registers for notification of changes to the *predicted* location, so that it can process them and fire signal strength prediction events in return. This, of course, requires that the context repository is queried with the purpose of finding historical context values with a reported location near the predicted point. This is also achieved with a context query, expressed as illustrated in listing 7.7.

This query uses the `SELECT` keyword to make a selection of all stored context values of the specified entity and scope (where the location of the device is represented with coordinates). The returned values are then filtered based on a conjunction of two constraints: their longitude and

[4]The functionality of using complex XML-based queries as input DMCs which is then transformed to Java code is not implemented yet. Rather, the developers are expected to specify simple queries in the form of an entity/scope pair.

latitude values are near the values “11.2” and “20.9” respectively. In this case, the *nearness* of the coordinates is computed via a delta value.[5]

```
<?xml version="1.0"?>

<ctxQuery resultName="nearbyLocation">

  <entity ontConcept="#Thing.Concept.EntityType.Device">this</entity>

  <scope ontConcept="#Thing.Concept.Scope.Location"
      ontRep="#Thing.Concept.Representation.Coordinates">
    location
  </scope>

  <action type="SELECT">
    <conds>
      <cond type="ONVALUE">
        <logical op="AND">
          <constraint par="location.#Longitude" op="EQ" value="11.2"
              delta="0.05"/>
          <constraint par="location.#Latitude" op="EQ" value="20.9"
              delta="0.05"/>
        </logical>
      </cond>
    </conds>
  </action>

</ctxQuery>
```

Listing 7.7: The context query used by the Signal strength predictor plug-in

With this query, the Signal strength predictor plug-in is able to retrieve all stored values of location context elements, which are close to the point of interest (i.e., the predicted coordinates for the upcoming location). Given this information, the plug-in identifies the time periods during which the device was present in that specific location, and by using them it queries the context repository again, this time to retrieve all signal-strength values for the corresponding time periods. By examining and prioritizing these values (i.e., dropping older values when multiple

[5]It should be noted that in the simulated scenarios that were used for testing, the values in the coordinates do not correspond to real values or scales.

measurements are available), the Signal strength predictor computes the estimated upcoming signal strength as the average of the past values. With this predicted value, the plug-in generates a new context event, which is communicated to the context middleware. Similar to the predicted location, the predicted signal strength context element is identified by an individual scope (i.e., `#Thing.Concept.Scope.NetworkPrediction`).

This example also shows one of the limitations of the CQL in its current implementation. In practice, the functionality of the plug-in consists of a selection of location points matching the desired (predicted) location. Then, it uses the timestamps associated to that location to search for context elements encoding the signal strength with a matching timestamp. In practice this realizes an indirect *inner join* operation, which in relational databases can be fully automated so that a single query can be used to implement the same functionality (while at the same time also minimizing the data transfer and optimizing the performance).

Finally, as the input context data used is imperfect (i.e., its accuracy ranges from zero to one), the Signal strength predictor plug-in should also evaluate these metadata and assess the accuracy of the predicted signal strength. The assessed accuracy is basically a function of two inputs: First the accuracy of the input location prediction and, second, the historical context values describing the signal strength at the predicted location. For instance, if the accuracy of the location prediction is high, but there are either too few, or too inconsistent or too old context data regarding that location, then the assessed accuracy of the signal strength prediction should be low. This information allows the end-users of the context information (e.g., the revised CaMP application) to make informative decision with regard to the predicted signal strength.

7.2.3 Deploying the Signal Strength Predictor

For the purpose of testing the correctness of the two plug-ins described in the previous subsections, a test-case was realized where the context repository was pre-loaded with a set of predefined context values. These values correspond to two context elements, encoding the sensed location and signal strength at specific timestamps. These values are illustrated in table 7.

Table 7: Set-up data for the Signal strength predictor case study

Timestamp offset	**Longitude**	**Latitude**	**Signal strength**
-1000	11.2	20.9	85%
-90	10.2	21.9	80%
-80	10.3	21.8	70%
-70	10.4	21.7	20%
-60	10.5	21.6	10%
-50	10.6	21.5	20%
-40	10.7	21.4	70%
-30	10.8	21.3	80%
-20	10.9	21.2	85%
-10	11.0	21.1	90%
0	11.1	21.0	95%

As illustrated in this table, the location is represented by two scalar values corresponding to Cartesian coordinates. Similarly, the signal strength is represented by a single scalar value in the range [0, 1] corresponding to a percentage value. Both context elements have a common timestamp metadata value. It should be noted that the longitude value increases linearly (with a step of 0.1) and the latitude decreases linearly with the same step. At the same time, the signal strength takes varying values, with a simulated *blind spot* at the location indicated by point [10.5, 21.6].

In the evaluation, the plug-ins were tested by installing them and activating them via an asynchronous context request for *predicted signal strength*. As mentioned already, the Location predictor plug-in is essentially a context sensor and the Signal strength predictor plug-in is a context reasoner. The simple architecture used to test these plug-ins is illustrated in figure 37.

In this figure, the Dummy context listener is used in the place of an actual context consumer (i.e., a context-aware application). For instance, the context viewer (see section 5.5) could be used to register to the middleware for the purpose of activating the plug-ins and testing their functionality.

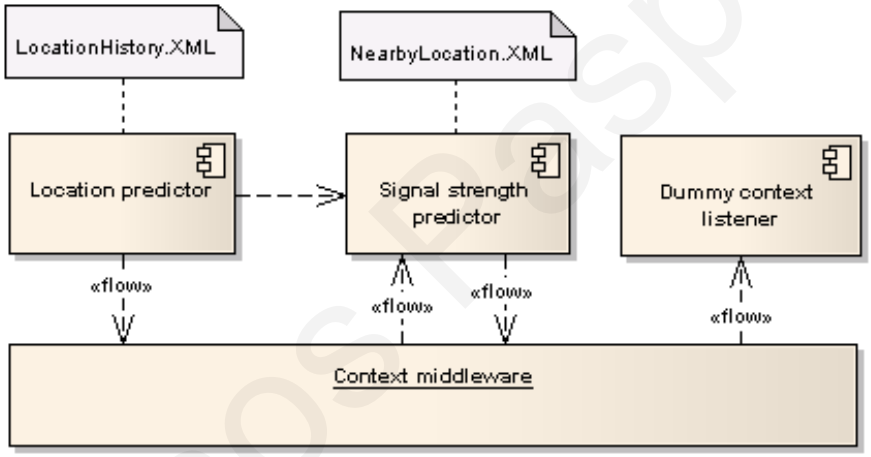

Figure 37: The architecture of the scenarios used to test the Signal strength predictor

During the case-study setup, the context repository is populated with the values shown in table 7, right before the plug-ins are activated. First, the Location predictor plug-in generates a context event encoding context information stating that the location is predicted to be [11.2, 20.9] in 10 seconds. This event is then delivered to the Signal strength predictor plug-in, which processes it as described in subsection 7.2.2. Checking the context repository for historical context values at that location, this plug-in infers that the signal is expected to have a strength of 85% at the new

location. In return, a new context event is created, encoding this prediction, and is delivered to the middleware for further processing and distribution.

While this is a simple scenario, featuring only a few (linear) data points, it is argued that it fulfills its main purpose which is to illustrate the use of the CQL and the context repository. Apparently a larger dataset or more complex algorithms could have been used to infer either of the future user location or future signal strength, but the interfacing with the middleware would remain the same.

7.3 User-based evaluation

Evaluating the effectiveness of a development methodology is a challenging task. This section aims at providing a quantitative analysis and evaluation of the development methodology and the middleware architecture. This is achieved by collecting and evaluating feedback received from developers who used both the methodology and the middleware to produce context-aware applications. The evaluation comprises two parts: First, the experience of developers working on the MUSIC pilot applications were collected and analyzed. Second, a number of undergraduate students were instructed the development methodology and the use of the middleware. Then, they were asked to use them to build context-aware applications as part of their lab assignments in a course on context-aware systems. Their feedback was collected through questionnaires, and analyzed to identify the strengths and weaknesses of the proposed approach.

While this form of evaluation was rather limited in terms of both the number of participants and the length of the questionnaire, it is nevertheless argued that it has achieved to provide valuable insight concerning the advantages and the limitations of the proposed methodology and the

supporting middleware architecture. Inevitably, the formality of this evaluation approach was limited as a result of the lack of standard evaluation methods in relation to context-aware applications, as it was also argued in [68].

7.3.1 Evaluation from the pilot developers

The first form of evaluation was performed with developers implementing pilot applications for the MUSIC middleware [6]. This middleware, which encapsulates the context manager as one of its core components, is used by the developers to create self-adaptive applications featuring several variation points, as discussed in chapter 2 and illustrated in figure 10. This kind of applications requires context data not only for direct use by the application, but also for allowing the middleware to select and apply the optimal application variant. In this regard, the developers of the pilot applications were often able to use the context middleware in a seamless way (i.e., by using basic context types provided by plug-ins that were already bundled with the MUSIC middleware, such as the resource sensors). In some cases, however, the pilot developers had to develop their own, custom plug-ins.

These developers had long programming experience and were familiar with OSGi-based middleware architectures. Their affiliation and the plug-ins they realized are listed in table 8.

The developers of these plug-ins were asked to compare the development of a context-aware application using the proposed model and middleware versus using an ad-hoc development approach. Two of them stated that they would use the proposed solution again if they needed to develop more context-aware applications, and the other two said they would use it again unless the targeted applications were too simple, in which case they would use an ad-hoc approach. All four of them agreed that the tasks for developing a plug-in were of low-to-medium complexity and that the hardest concept was the design of the context model.

While this evaluation indicated that the proposed middleware architecture was favorably accepted by developers with minimal exposure to the development methodology, another survey was attempted, this time in the more controlled setting of an undergraduate course.

Table 8: The plug-ins developed by the pilot-application developers

Partner	**Plug-in**	**Description**
Telefonica I+D	*Radio-Frequency Identification* (RFID) sensor	A plug-in that monitors an underlying RFID hardware sensor and generates an event when an RFID tag is discovered. This plug-in is used in the TravelAssistant pilot application.
HP Italy	Resource sensor	A multi-sensor bundle providing information on the availability of the following resources: battery, memory, microphone, network, screen and speaker. This plug-in is used in all the pilot applications of MUSIC.
NTNU	Resource utilization sensor (simulated)	A simulated sensor which has a GUI-based control to simulate context events characterizing a customized context type. This type represents resource optimization in the device. The plug-in was used in the realization of the InstantSocial pilot application.
Integrasys	Resource sensor (simulated)	A simulated resource sensor plug-in developed for enabling quick testing of the adaptation behavior of the SatMotion pilot application.

7.3.2 Classroom-based evaluation

As part of an undergraduate course at the University of Cyprus (EPL-429, Spring 2009), a number of participating students were asked to implement context-aware applications by following the methodology presented in this thesis and by using the provided middleware framework. These applications were constructed by realizing the business-logic of the application and by producing new, or reusing existing, context plug-ins as per the proposed methodology. With these, the students formed and deployed their applications by integrating the appropriate components (i.e., plug-ins and business logic) with the middleware. Finally, they presented their applications and

provided their feedback by answering some questionnaires. The aim of those questionnaires was to evaluate the strengths and weaknesses of the methodology and the middleware.

As the students had limited programming experience and zero experience with OSGi, they were first provided a quick introduction to OSGi (and in particular of its Equinox implementation). Next, they were given an introduction to the middleware architecture and a tutorial on how to develop context plug-ins using the proposed methodology. Eventually, they were asked to complete a practical lab assignment, spanning three phases:

- Describe a context-aware application: specify its business and context-aware logic, and identify the required context types and the corresponding plug-ins.
- Realize the context plug-ins: develop some plug-ins from scratch and also reuse some existing ones (perhaps developed by a different team).
- Implement the application's business logic to use the context data provided by the developed (or reused) plug-ins. Deploy all of them on the middleware.

The name and the description of the resulting applications, as well as the context plug-ins they developed or reused, are summarized in table 9.

This structured approach has helped teach the students the basics of context-aware applications, as well as the benefits of component-orientation. By getting hands-on experience with the development of context-aware applications, they were able to better grasp the complexity inherent in their development. Also, by reusing existing context plug-ins, they were able to experience the benefits of COTS-like development.

Table 9: The applications developed as part of the EPL-429 course

Application name	**Description**	**Developed plug-ins**	**Reused plug-ins**
Automatic attendance recorder	Automatically detects which students attend a course lecture (based on their Bluetooth devices), and generates a report at the end of the lecture which includes the lecture description based on the course's calendar.	Bluetooth sensor	Calendar sensor
Presentation buddy	Based on the location of the user, this application checks the user's calendar and if a lecture is scheduled in the following minutes, then it automatically launches the appropriate presentation.	Calendar sensor	Location sensor
Conversation recorder	Automatically starts voice recording when the noise level exceeds a threshold; also uses information from the Bluetooth sensor to tag the conversation (e.g., IDs of nearby devices).	Noise sensor	Bluetooth sensor
Points-of-interest guide	Based on the user location (e.g., current city), this application presents a list of points-of-interest, and when the network connectivity is available, also their Wikipedia description.	Location sensor (based on the IP address), Connectivity sensor	-
City buddy	It includes a predefined list of points-of-interest, and based on the user's location and upcoming events, it suggests activities (e.g., specific restaurants to go for lunch, etc)	Calendar sensor	Location sensor

Once the assignments were completed, the students were asked to fill-in an anonymous survey, providing input about the advantages and disadvantages of the development methodology and the middleware architecture they used. The questions in this survey, along with the summary of the answers they provided, are listed in appendix B.

7.3.2.1 Survey results

The survey includes five sections. The first one (i.e., B.1) provides a summary of the participants background. The next two (i.e., B.2 and B.3) compare the proposed methodology with ad-hoc approaches and were answered by all the participants. The last two (i.e., B.4 and B.5) were specific to the MDD-based approach—described in chapter 6—and were thus answered only by a subset of students who used the approach.

It should be noted that the participants did not consider a point-of-reference (e.g., a competing development platform) when they evaluated the approach proposed in this thesis. Rather, their feedback compares the proposed approach with ad-hoc development methods.

The feedback collected by the students has shown that:

- Almost all of the students (11 of 12) would prefer to use the proposed development approach and the provided middleware—either partially or completely—rather than an ad-hoc approach, if they needed to develop more context-aware applications.
- Most tasks related to the development of the plug-ins were of low-to-medium complexity (except the task of creating the context model which was of medium-to-high complexity).
- On average the (teams of) students spent approximately 23 hours in preparation (i.e., studying the material and the examples) and another 20 hours for the code creation (of both the plug-ins and the business logic of the application), signifying a rather smooth learning curve.

A subset of the students were also guided to re-implement the same plug-in—that they had developed manually—a second time, by using the proposed MDD-based approach. Because of time constraints and also because the students had minimum experience with modeling and transformation tools, they implemented the model of their plug-ins and transformed it into code under supervision.

Regarding the comparison between the manual and the MDD-based approach, the feedback documented in the questionnaires has shown that:

- The MDD-based approach is preferred to the manual approach, given that the developer is familiar with the tools (i.e., the MDD-based approach is ideal when the developers need to

develop context plug-ins on a regular basis). In particular, the participants liked the fact that they could visually interact with, and form the conceptual model of the plug-in.

- The modeling approach is also preferred to the manual approach as it enables the developers to easily produce bug-free code (especially the code realizing the context model). On the other hand, this approach was found to limit the access to the (automatically generated) implementation code, something that can cause experienced developers to feel restricted.

It should be noted that the participating students had limited experience in Java and no experience in either of the OSGi framework or the Enterprise Architect modeling tool. It can be assumed that had the developers been more familiar with these concepts, they could have experienced an even smoother learning curve, resulting to spending less time on preparation leaving more time for familiarization with the middleware and the development steps.

7.4 Requirement-driven evaluation

This section evaluates the requirements identified in section 3.2 (and summarized in table 1), by revisiting them and evaluating the proposed development methodology and middleware architecture against them. It should be pointed that as many alternative implementations are possible—depending on assumptions about how context information is queried, interpreted, etc—it is not feasible to present a straightforward and extensive quantitative comparison (beyond, for instance, the one presented in the previous section). Therefore, the analysis in this section is mostly qualitative rather than quantitative, except where feasible.

The following paragraphs discuss the list of functional and extra-functional requirements individually, arguing to which extend has each requirement been addressed in the current state of the development methodology and the middleware architecture. In some cases, directions for improvement are also proposed .

7.4.1 Evaluation of the functional requirements

The functional requirements are those which deal directly with features explicitly related to the scope of context-awareness.

7.4.1.1 Application specific context acquisition, analysis and detection

This fundamental requirement is addressed by the middleware architecture which offers a dedicated context access service (see subsection 5.3.1). The context-aware applications can simply request or register for their desired context information types and let the middleware attend to the tasks of acquisition, analysis and triggering. Furthermore, the context access service also supports a rich context query language which allows context filtering based on predefined conditions. This empowers the developers to specify complex context queries rather than implementing complex context filtering logic inside their applications.

7.4.1.2 Context triggered action

Context triggered actions are necessary for the development of efficient context-aware logic. By adopting the classic publish-subscribe pattern [52], the developers can register their code for asynchronous notification of relevant context changes without having to explicitly inquire it periodically. In the proposed middleware architecture, this requirement is addressed by the context access service (see subsection 5.3.1), which allows for asynchronous context queries. For example, the CaMP application uses the context access service to subscribe to changes in the context type corresponding to the user's presence in the room (see subsection 7.1.6).

7.4.1.3 Heterogeneity

Heterogeneity of context information is a major hurdle in interoperability. Not only is it required for enabling context distribution, but it is also a major requirement for enabling the formation of context-aware applications constructed out of individually developed components (i.e., third-party context providers). The layered context model presented in section 4.4 faces the challenges of context heterogeneity by allowing ontology-backed inter-representation operations (see subsection 4.4.3). Naturally this method heavily depends on a common context model, which in the default implementation described in this thesis is backed by an ontology.

7.4.1.4 Uncertainty

Because of the imperfect context sensing methods and mechanisms used in the production of context-aware applications, the developers need to consider and compensate for the uncertainty which characterizes context information. The proposed methodology partly supports this by adapting a flexible context metadata model, allowing for arbitrary metadata types (see subsection 4.4.2). For instance, the developers can specify an *accuracy* metadata type, which characterizes the generated context type, and which can be further considered during the evolution of the context information through the hierarchical decision chain (see figure 10). For example, the accuracy characterizing the *location* context type values—reported by the Location predictor plug-in—is further assessed by the Signal strength predictor, which computes a combined accuracy and attaches it to the *signal strength prediction* values. At the end of the decision chain, the context-aware application can use this information to decide whether the accuracy is sufficiently high to justify an action (e.g., resume or suspend the media playback in the CaMP application).

7.4.1.5 Context histories

This functional requirement is fundamental for enabling advanced context use. Providing access to historical context information is important because it enables advanced context reasoning methods, such as location prediction and signal strength prediction discussed in the context of the SSP case study (see section 7.2). The proposed middleware architecture enables storing historical context data through a dedicated context repository service. Past context data is accessed via the context access service. Context data corresponding to specific time periods can be accessed by specifying the appropriate conditions in CQL (i.e., via the *timestamp* metadata type). Such a query was illustrated in the SSP case-study where the Location predictor plug-in defines a query where values within a given timestamp range are requested (see listing 7.6).

7.4.1.6 Inference of interdependencies of context

This requirement refers to the ability of applications to infer interdependencies (i.e., relationships) between context types. The context model presented in section 4.4 currently provides no support for this sort of inference. However, in ontology-backed context models, it is possible to provide such support by exploiting functionality from the underlying OWL functionality [13].

7.4.1.7 Support for multiple concurrent applications

The support for multiple concurrent, context-aware applications is an important requirement, especially as modern mobile devices are powerful enough and feature multi-task operating systems (such as the Android platform [11]). Both the development methodology and the middleware architecture were designed by carefully considering this requirement. The resulting methodology and the middleware architecture split the context-aware applications into context providers (i.e., context plug-ins) and context consumers (i.e., applications). The middleware acts as an intelligent layer connecting the context providers to the consumers in a seamless, and dynamically adaptable

manner. In theory, any number of context providers can be bound to the middleware, servicing any number of context-aware applications. In practice, these numbers are limited by the actual resource constraints of the device, but in this case the middleware is not the bottleneck (see discussion on scalability further on in sub-subsection 7.4.2.7).

7.4.1.8 Transparent distribution of context

Transparent distribution of context is achieved in two ways. First, the context model allows for globally-valid, unambiguous reference to context information. Transformation from and to locally-valid and globally-valid context information (from both a semantic and a representational point-of-view) is automatically undertaken by the context model, as it was discussed in subsection 4.4.4. Second, the middleware architecture provides functionality that allows third-party components to inquire the locally provided and required context types. This allows the implementation of context distribution systems which access this information, and use it to form an extended, federated context space. As discussed in section 5.4, a SIP-based implementation of such context distribution system is already implemented in the context of the MUSIC project [6, 21, 43].

7.4.1.9 Privacy of context information

Privacy of context information refers to the requirement of protecting the identity of the user, along with his private context information. Privacy concerns are raised in the case of context distribution, and also in the case a mobile device is stolen or lost. As the context distribution system is treated in this thesis as an external component, the main effort is placed on protecting the privacy in the latter scenario. In this case, the main step towards user privacy protection would be the realization of a password-protected, context repository system with data encryption, preventing unauthorized access to the user's data. However, the implementation of such a system was beyond the scope of the work in this thesis.

7.4.1.10 Traceability and control

The original scope of this requirement was to allow the end-users understand the behavior of the context-aware applications, enabling them to manually intervene when needed. The rationale for this functionality is to allow the end users to trust the autonomous context-aware logic, and thus adopt the technology. However, the development methodology and the accompanying middleware architecture proposed in this thesis primarily aim at facilitating the developers into designing and implementing complex context-aware applications easier and more efficiently. The actual context-aware behavior, as well as the end-user's ability to control it, still remains largely in the hands of the developers. However, limited support for traceability is provided, either through specialized plug-in viewers (as shown in figure 35), or via the use of a viewer component (see section 5.5). In both cases, the user views the context values at run-time. It should be noted however, that the context viewer was primarily designed for simulation and testing purposes, and is not really intended for use by the end-users, although custom context monitors and controllers are of course supported, as illustrated in the CaMP case-study (see subsection 7.1.7).

7.4.1.11 Interoperability and standards

The last functional requirement is that of interoperability and standards. While the domain of context-aware applications remains rather limited, no standard has achieved to establish itself as a *de facto* protocol for context distribution and dissemination yet. However, the author of this thesis envisions that such a standard might not be too far in the future, although privacy and security concerns might still limit its applicability. In the meantime, it is argued that building the middleware architecture on top of a quickly proliferating component framework standard, such as the OSGi, is a significant advantage because it greatly facilitates a smoother learning curve

for adopters of the methodology and the middleware, while also taking advantage of a robust codebase.

7.4.2 Evaluation of the extra-functional requirements

The extra-functional requirements are those which deal with general-scope features, not necessarily explicit to context-awareness. While some of these requirements were greatly facilitated by the underlying OSGi component framework, others required much more elaborate handling.

7.4.2.1 Ease of building

One of the main goals of this thesis was to allow building context-aware applications in an easy, and efficient manner. For this purpose, it was decided that enabling code reuse and development with separation of concerns would greatly facilitate this goal. While it is difficult to claim that the proposed methodology supported by the middleware architecture fulfill this requirement completely, it is argued that the current indications are quite positive. As argued at the beginning of this section, performing a quantitative rather than qualitative evaluation of this requirement is not always feasible (mainly because there is a lack of standard evaluation methods and guidelines for context-aware applications [68]). In this respect, a quantitative evaluation was attempted by using questionnaires, as it was discussed in section 7.3. The results from these were quite positive but, naturally, in absolute terms rather than in comparison with other relevant methodologies). On the other hand, a qualitative approach of the ease of building is indirectly achieved by evaluating requirements which are known to contribute to better and easier software development methods, such as the extra-functional requirements discussed in the following paragraphs.

7.4.2.2 Modularity

The requirement of modularity was one of the main driving forces during the design of the middleware architecture. Allowing an architecture that can be easily configured to match the needs and constraints of different platforms is an important advantage, especially in the context of mobile and ubiquitous computing which are characterized by high heterogeneity and variability. For instance, the modularity of the architecture presented in section 5.3 enables different configuration of the middleware, both lighter for resource-constrained devices and more powerful for demanding applications. For example, a context-aware application with no need for context distribution, historical context values or complex context queries can be deployed on a lightweight configuration of the middleware, which includes a plain realization of the context repository service only. It should be noted that creating different variants of the middleware architecture can be achieved both at design-time (when new, specialized plug-ins are needed) and at deployment-time quite easily. The developers need only install the corresponding bundles, containing the required middleware components. Then, the OSGi framework, with help from the *Declarative Services* run-time, handles their binding automatically.

7.4.2.3 Separation of concerns

The separation of concerns is a popular method for easing the complexity of a problem by breaking it down to smaller, simpler pieces. Furthermore, when the individual pieces are independent, then they can be developed in parallel by individual developers. With this rationale, the development methodology described in chapter 4 separates the development of context-aware applications into the tasks of developing the context providers and the context consumers. The former are mostly realized as context plug-in components and are highly reusable. The latter are typically context-aware applications or middleware components and are only loosely coupled

with the context providers. This approach has also the advantage that the applications can be more easily tested and debugged, because these tasks are performed at the level of the individual components.

7.4.2.4 Code reuse

Enabling code reuse was one of the main goals of the proposed middleware architecture (see chapter 5). Adopting a model which treats the context providers as independent, pluggable components, greatly facilitates this goal. The developed components are treated as black-boxes, where their internal functionality is hidden and only their context offerings and context requirements are explicitly defined (as metadata). These metadata are also used for publishing the plug-ins in component repositories (see subsection 4.3.1), further facilitating code reuse.

7.4.2.5 Uniform development support

Establishing uniform development support for context-aware applications was beyond the scope of this thesis and has not been addressed.

7.4.2.6 Evolution

Software evolution of context-aware applications is greatly facilitated by the fact that applications are split into loosely coupled context providers and context consumers. For instance, a context-aware application can easily update its context-aware logic by replacing some context providers with updated ones, delivering either better quality of context information, or better resource utilization (or both). Similarly, evolving the business logic of the application can be achieved without requiring changes to the context providers.

7.4.2.7 Scalability

Scalability refers to the ability of an architecture to gracefully accommodate an increasing number of components, both locally and remotely. With regard to context distribution, it was already mentioned that the design of an appropriate, possibly scalable, distribution system is beyond the scope of this architecture. With regard to local scalability (i.e., in terms of the number of context provider and context consumer components), it is argued that the middleware architecture offers a bottleneck-free path for deploying context-aware applications. Supported by the underlying OSGi component framework, numerous components can be deployed and handled, constrained only by the resources and the capabilities of the deployment platform. The resolution mechanism is triggered only when a new plug-in is installed or an existing one uninstalled, while the activation algorithm is used only when new applications are started or existing ones are stopped. Besides the fact that both mechanisms are triggered infrequently, they are both completed in polynomial time (as a function of the number of the context providers and context consumers). Thus, from the perspective of local-deployment, the middleware architecture and its mechanisms are both highly scalable.

7.4.2.8 Dynamic behavior

Dynamic behavior is enabled by allowing new context plug-ins to be installed and activated at run-time. As the context providers and the context consumers are only loosely coupled, it is possible to have applications replace their context-aware logic at run-time in a seamless manner. This is naturally supported by the underlying OSGi framework, which allows for dynamic installation—or un-installation—and activation—or deactivation—of components, packaged in bundles. This feature is important as it allows mobile context-aware applications to take advantage of richer

context information when it becomes available, and rolling back to basic context data use when it becomes unavailable.

7.4.2.9 Platform independence

Platform independence is important because, as it was argued earlier, mobile and pervasive computing environments are characterized by high heterogeneity and variability. In the proposed approach, platform independence is achieved in multiple ways. First, building on the underlying framework's properties (i.e., Java-based, OSGi compliant), the middleware architecture implementation is highly reusable across different platforms. Furthermore, high-level context reasoner components—which are usually closely related to specific context-aware applications—are also highly platform independent as they are designed to be OSGi compliant. On the other hand, low-level context sensors are often highly dependent on specific platform properties (e.g., on hardware or OS libraries). Nevertheless, these components offer context types which are commonly used by multiple applications, thus justifying individual developments for various platforms. A more elaborate discussion of the implications of the hierarchical decision chain on platform independence can be found in subsection 4.2.4.

7.4.2.10 Lightweight architecture

This requirement is quite relevant to the *modularity* requirement discussed earlier in subsubsection 7.4.2.2. Supported by its modular design, the middleware architecture provides a lightweight framework, capable of realizing context-aware applications with a small footprint. The most lightweight configuration of the middleware (as of version 0.4.0) is sized under 165 Kb.

7.4.2.11 Adoption of existing patterns and standards

The proposed development methodology and the middleware architecture build on existing patterns and standards when possible. For instance, the middleware architecture uses the common publish-subscribe pattern to enable asynchronous context access. Furthermore, the middleware architecture builds heavily on top of the OSGi standard for modularity, as it was discussed in section 5.6.

7.4.2.12 Support for mobility

The requirement of mobility was one of the core requirements that lead to the design of the context middleware. The selection of OSGi R4 as the underlying component framework is in-line with this requirement, as OSGi was also designed with mobile and embedded computing in mind. Furthermore, because of the limited resources characterizing mobile devices, it is also important that the developed middleware is *efficient* in its resource utilization (see sub-subsection 7.4.2.15).

7.4.2.13 Fault tolerance

Fault tolerance is an important feature, aiming to guarantee that the system is able to overcome faults that are limited to specific parts of the software or the hardware. In the case of distributed scenarios, faults often occur at the network level, thus requiring that the underlying mechanisms are capable of overcoming them. On the other hand, at a local level it is possible that the code implementing a context provider plug-in halts either because of a hardware problem in the underlying sensor, or a software problem (e.g., a bug) in its code. The middleware architecture presented in this thesis does not explicitly deal with these requirements. With respect to fault-tolerance in distributed scenarios, it is expected that the corresponding distribution mechanisms will handle this requirement. On the other hand, fault tolerance at the plug-in level can be enabled in a relatively straightforward manner. In the current implementation, the plug-ins are driven by a

custom thread-pool which periodically invokes methods in the plug-ins as needed. By improving this thread-pool to also monitor the individual threads, and detect unjustified delays, it is possible to allow for automatic isolation of faulty plug-ins, while also notifying the middleware and the end-user of the situation.

7.4.2.14 Ease of deployment and configuration

The deployment and configuration of context-aware applications can be challenging for non-experts, especially when specialized hardware and software is used. From the developers' perspective, the fact that the middleware and the context-aware applications are developed and packaged as OSGi bundles makes it an easier task for the end-users, assuming they are familiar with OSGi and its implementation-specific management GUIs.[6] From the end-user perspective, the ease of deployment and configuration remains a challenge that must be met by the individual developers of the context-aware applications, depending on the actual software and hardware required by them.

7.4.2.15 Resource efficiency

The final extra-functional requirement is that of resource efficiency. This is achieved in two ways: First, by adopting a modular, lightweight architecture which minimizes the resource consumed by itself. Second, by using an intelligent mechanism to activate and deactivate the context plug-ins as needed (see subsection 5.2.4). In subsection 5.7.2, it was shown that the use of this mechanism produces significant improvements to the resource consumption.

[6] For instance, Knopflerfish offers a user-friendly GUI which can be used to monitor and control the available OSGi bundles

7.5 Conclusions and future work

This chapter has presented two case-study applications, demonstrating the use and showing the benefits of the development methodology described in chapter 4 and the middleware architecture described in chapter 5. Furthermore, it has described the evaluation process that was followed and which included both quantitative and qualitative analysis.

The first case-study application—the Context-aware Media Player—was used to demonstrate the development methodology and also to showcase how context plug-ins can be developed either manually or with the use of MDD. Furthermore, the CaMP case-study has demonstrated the convenience of software reuse by explaining how the developers can use a component repository and reuse a suitable component rather than designing and implementing a new one. The section concluded with a discussion on the integration process and by showing how the application is deployed and run.

The second case-study application presented in this chapter—the Signal Strength Predictor—is a simple application used to illustrate the use of the CQL. This application demonstrated how the plug-ins can be easily defined with the use of CQL-based queries. Also, SSP has demonstrated how the MDD-based approach supports the design of plug-ins using CQL-based conditions for filtering the input context, while at the same time reusing existing operator components—such as the Value predictor operator—to realize context prediction functionality.

While the two case studies demonstrated the use of the proposed methodology and the middleware architecture, this chapter also aimed at evaluating them.

A quantitative evaluation, which was necessarily limited by the lack of standard evaluation methods and guidelines in relation to the development of context-aware applications, was presented. Primarily aiming to measure the ability of the methodology and the middleware architecture to facilitate the development of context-aware applications, an experimental approach was followed. In this approach, developers used the methodology, the development tools and the middleware architecture hands-on, in order to create either individual context plug-ins or complete context-aware applications. Their input was eventually collected with anonymous questionnaires, and analyzed to infer the strengths and limitations of the development approach.

Finally, this chapter has also presented a qualitative evaluation of the development approach. For this purpose, the extensive set of requirements detected in chapter 3 and summarized in table 1 were revisited, and the development methodology and middleware architecture were individually evaluated for each requirement. While the evaluation has already indicated that the development methodology and the middleware add significant value in the hands of developers creating general context-aware applications, a number of limitations were also identified. The main limitation is that although these tools are useful and preferable to the ad-hoc approach, they still add (significant) complexity compared to the latter. For instance, the development of the model supporting the context plug-ins and the context-aware applications was found to be challenging for the developers.

Naturally, the evaluation process described in this thesis has also resulted to some pointers for future work. For instance, the context model is believed to be one of the most complex and challenging aspects of the methodology, rendering its improvement a high priority for future work. For example, the model could be refactored so that entry-level developers can use just a simple subset of its features, advancing to the more complex features (e.g., inter-representation transformation and CQL-based queries) as needed. This would contribute to an even smoother learning curve,

without degrading the potential power of the model. Another improvement concerns the use of MDD-based tools, but only for the development of the context model, rather than the complete architecture of the context plug-in. This would allow the exploitation of the graphical, UML-based tools for designing and realizing the (extended) context-model of an application without requiring that the developers become familiar with the complete range of models and tools needed for developing full context plug-ins.

Chapter 8

Conclusions

This chapter concludes the thesis by summarizing its research contributions and by providing a discussion of key topics for future work.

8.1 Summary of contributions

The primary goal of this thesis was to provide software engineering support for the development of context-aware applications. In this respect, a methodology was proposed, allowing the developers to efficiently create context-aware applications by using separation of concerns while also facilitating software reuse. Furthermore, a pluggable and modular middleware architecture was presented, enabling the deployment of the developed applications. It was shown that this architecture features multiple benefits, including platform independence and optimization of resource usage.

The proposed methodology and the middleware architecture differ from related approaches, such as those described in chapter 3, because they were designed with the aim of being comprehensive, allowing the development of arbitrary context-aware applications. As the same time, a broad set of requirements was identified and used to guide the design and development of the methodology and the middleware, resulting in a highly dynamic and modular architecture. For instance

unlike the current state-of-the-art, the proposed middleware implements a pluggable architecture which allows the applications to take advantage of richer or more efficient context providers as they dynamically become available while moving about in space.

The research contributions of this thesis are summarized, by chapter, in the remainder of this section.

Chapter 2 presented the foundations based on which the thesis forms the development methodology and the middleware architecture. This chapter first contextualized the work of this thesis by providing definitions for concepts such as context, adaptation and utility. Based on these definitions, a conceptual model of context-aware, self-adaptation enabling middleware was presented. This model represents a greater vision for a middleware architecture that enables the development of not only context-aware but also self-adaptive applications. The results of this thesis cover primarily the context-management aspect of this vision. Nevertheless, further work is planned to extend these results to also handle the variability of applications (see section 8.2).

Chapter 3 presented a survey of related work. This survey examined multiple approaches—presented in a chronological order—achieving the following results: First, it provided insight into the community's evolving view of the problem of context-awareness by studying the major challenges identified by researchers in the area. Second, it summarized an extensive list of requirements for context-awareness enabling frameworks. These requirements were classified into functional, reflecting context-specific features of the frameworks, and non-functional, describing more general system properties. Finally, the same chapter presented a survey of different approaches for developing context-aware applications, along with a discussion of their advantages and limitations.

Chapter 4 introduced the development methodology. It first presented a method where the context-aware applications are designed as individual context providing and context consuming

components, thus enabling the development of such applications with separation of concerns. Second, it presented a methodology describing the steps followed during the development of context-aware applications using this method. Finally, the development methodology was complemented by a context model and a context query language.

The development approach builds on the well-known advantages of the separation of concerns method. By separating the context-aware application into context providing components (i.e., context sensing and reasoning plug-ins) and context consuming ones (i.e., the business-logic components of the applications) the development task is simplified. Furthermore, by conforming to the well-defined specifications of context plug-ins, it facilitates the formation of component repositories thus promoting their reuse across different applications.

The context model is designed as a generic service offering the ability to provide semantic consistency across different domains, while also facilitating inter-representation transformation. An ontology-backed implementation of the context model is described in more detail, along with an accompanying context query language allowing elaborate selection of context data based on specified conditions.

Chapter 5 presented a middleware architecture which allows the deployment of applications developed using the proposed methodology. Unlike existing approaches, this architecture features two important properties: It allows dynamic activation of context sensors and reasoners through a pluggable architecture, and it enables easy customization to various platform characteristics.

The pluggable architecture allows the construction of context-aware applications where the context providers are not determined *a priori*. Rather, the applications only specify their context needs and, based on them, the middleware seamlessly binds them to dynamically available context providers. At the same time, the middleware monitors the dynamically changing context needs of the applications, and activates or deactivates the context providers accordingly. As it was shown

in this chapter, this approach improves the resource utilization of the middleware, which is an important advantage for resource-constrained devices.

Furthermore, it was shown that the modular architecture of the middleware allows the developers to configure it—or extend it—so that it better fits the constraints of the targeted environments. For instance, when the targeted platform is sufficiently powerful, a database-backed context repository can be used to realize the context history functionality. The modularity of the middleware builds on the capabilities inherited by the underlying OSGi framework.

Chapter 6 presented a model-driven development approach for the creation of context plug-ins. This approach includes a conceptual model for the design of the plug-ins using predefined artifacts. It was argued that the use of this MDD-based approach extends software reusability from complete plug-in components down to sub-component artifacts such as data-structures, operators and connectors. Unlike other MDD-based approaches for context-aware applications, the proposed technique is geared towards the development of context sensing and context reasoning components, which are constructed of reusable software artifacts and become reusable themselves.

Finally, chapter 7 described the evaluation process followed in this thesis. The use of the development methodology and the middleware architecture was first illustrated in the context of two case-study applications. Then, the results of an experimental evaluation process were presented, followed by an analysis of the conformance of the proposed solution to the requirements identified in chapter 3.

The first case-study application presented was the Context-aware Media Player. This application has demonstrated the development methodology by describing the steps followed while developing an application utilizing three context plug-ins. This case-study has also illustrated how the plug-ins are seamlessly interfaced by the middleware in a loosely-coupled manner, realizing the context-aware logic of the application. The second case-study application presented

was the Signal Strength Predictor. This application has primarily demonstrated the use of the context query language in the context of two plug-ins. It has also shown how the use of predefined, generic operators can be used to realize functionality such as context prediction, using the proposed MDD-based approach.

The experimental evaluation presented in chapter 7 consists of two parts. First, the developers of context plug-ins were asked to provide their feedback with regard to their experience with the methodology and the middleware. Second, the students of an undergraduate course used the development methodology and the middleware to develop context-aware applications as part of their lab assignments. Their experience was then collected through a survey and analyzed, indicating that the proposed solution adds significant value in the hands of developers of context-aware applications. Finally, this chapter also presented a qualitative evaluation. This evaluation revisited the functional and extra-functional requirements identified in chapter 3 and showed that the proposed solution has achieved to address, to a great extend, the majority of them.

8.2 Future work

A conceptual model for enabling a wider domain of context-aware, self-adaptive applications was presented earlier in chapter 2. This model was based on a general middleware architecture, enabling component-based applications that are able to automatically and autonomously adapt to dynamic changes in the environment. While enabling this kind of applications is of high importance, its realization is particularly challenging, requiring improvements in several research directions. This thesis has presented results primarily in the area of *context management*, while the *component management* is largely provided by the underlying OSGi framework (see figure 6). Nevertheless, the *composition* and *adaptation management* aspects were covered only to a limited extend.

The concluding sections of chapters 4, 5 and 7 have identified a number of possible directions for future work, towards extending the original research results presented in this thesis. This section presents two additional directions for future work, spanning in the wider area of context-aware, self-adaptive systems. These are discussed in the following two subsections.

8.2.1 Developing context-aware adaptive applications with reusable components

This thesis has provided research results primarily in the area of context management. Nevertheless, realizing the vision of ubiquitous computing requires advances in additional areas. For instance, the high variability which characterizes mobile and pervasive computing environments highlights the importance of adaptation as a means for dynamically adjusting the software in order to better utilize the available infrastructure.

As it was discussed in [90], component-orientation is one of the fundamental technologies required for enabling software adaptation (see also section 2.4). Component orientation provides significant advantages, such as modularization and software reuse. However, general software componentry techniques deal mainly with the business aspects of components (i.e., their core functionality). At the same time, reusability of components is primarily done at the functional level only.

It is argued that extending the classic component-orientation model can greatly improve the task of developing context-aware adaptive applications. Such extensions would allow, for instance, the development of components in different phases, dealing with just one of the relevant aspects at a time. An initial attempt towards this direction was presented by the author in [105]. This publication provides a method for simplifying the development of context-aware, self-adaptive systems by separating their development into the concern of specifying their functional behavior, and the concerns of realizing where, when and how the application adapts.

Regarding component reusability, it is argued that an important direction for future work is to enable the reusability of components at both the functional and extra-functional levels. The original motivation for component orientation was the provision of reusable functionality, available through well-defined interfaces. However, in specialized scenarios, as in the case of context-aware adaptive applications, it is important that component reusability is extended to cover their context-awareness and self-adaptation properties.

For example, consider a component realizing text-to-speech functionality. Besides its functional behavior, the component also features specific extra-functional properties, such as different modes of operation (e.g., producing different levels of speech-quality depending on the available resources). While the reusability of the core functionality of such components has been extensively studied and is considered an established practice, allowing the reuse of the component at an extra-functional level is still an open problem. In this case, it would be beneficial to specify metadata in the component describing its extra-functional properties in a way that would enable their use in various component compositions, potentially by different applications, at run-time.

While this thesis presented a methodology facilitating the reuse of context-plug-in components, enabling the reusability of general-purpose components at both the functional and extra-functional levels remains an open problem.

8.2.2 Learning from user-feedback for improving the context-aware behavior

Part of the vision of ubiquitous computing is about enabling autonomic behavior in applications, so that the interaction required by the users is minimized. However, although the users could enjoy reduced interaction, they are often reluctant to trust machines making decisions for them. It is actually believed that their reluctance to have the control taken from their hands is one of the main hurdles preventing widespread adoption of context-aware, self-adaptive systems [112, 142].

An approach for overcoming this problem is to provide transparent adaptation algorithms, allowing the users to oversee the decision process and intervene to change a decision when it does not match their choice. For example, in the case of the Context-aware Media Player application presented in section 7.1, the users have the control to pause or resume the media playback, while at the same time they can always examine the *under-the-hood* tab to monitor the sensed context values and get some insight regarding the selected mode.

Here, it is argued that extending this model to also monitor the user-feedback, as they respond to the autonomously-taken decisions, has high research value. The reason is that the collected user-input can be stored and analyzed, perhaps using machine-learning algorithms, allowing the context-aware adaptation logic to optimize itself at run-time.

An initial step towards this direction was presented by Kakousis and the author in [80]. This paper described a method where the user was informed (via a non-intrusive message) whenever an adaptation decision was taken as a result of context changes. The user had the option to intervene and reject the decision, selecting an alternative one (e.g., maintaining the previous state). This feedback was collected and used for adjusting the decision algorithm and was thus taken into consideration in future decisions. However, this work has only presented some initial results which apply to limited-domain scenarios. Allowing user-intervention and dynamic optimization of the context-aware, self-adaptive behavior, perhaps by using machine-learning techniques, remains largely an open issue and a promising direction.

Appendix A

Source code of the core services of the middleware

This chapter lists selected parts of the source code realizing the prototype implementation of the context middleware. This implementation was performed in the context of the MUSIC project [6]. Certain parts of the source code (like the copyright declaration and the `import` statements) were removed to avoid cluttering.

The following sections describe the interfaces used to specify the services *provided* by the context middleware, as well as those *required* by it. These are the fundamental interfaces for explaining how the middleware is composed of sub-components, thus realizing a flexible, modular architecture.

A.1 Provided services

```
/*
 * The MUSIC project (Contract No. IST-035166) is an Integrated Project (IP)
 * within the 6th Framework Programme, Priority 2.5.5 (Software and Services).
 */

package org.istmusic.mw.context;

import org.istmusic.mw.context.cqp.queryapi.IContextQuery;
import org.istmusic.mw.context.events.IContextListener;
import org.istmusic.mw.context.exceptions.ContextException;
import org.istmusic.mw.context.model.api.*;
```

```
/**
 * This is the main point of interaction with the context middleware. Any
 * context client requiring access to the context information uses this
 * interface to achieve it. Various methods are provided to access the context
 * data either synchronously or asynchronously and with or without the help of
 * the Context Query Language.
 */
public interface IContextAccess
{
    /**
     * Provides synchronous access to the specified context data.
     *
     * @param entity describes the entity of the desired context data
     * @param scope describes the scope of the desired context data
     * @return a dataset containing the context which corresponds to the
     * requested entity/scope
     * @throws ContextException when an error occurs
     */
    public IContextDataset queryContext(
            final IEntity entity,
            final IScope scope)
            throws ContextException;

    /**
     * Provides synchronous access to the specified context data. It returns
     * the last context element stored only.
     *
     * @param entity describes the entity of the desired context element
     * @param scope describes the scope of the desired context element
     * @return the {@link IContextElement} which corresponds to the requested
     * entity/scope, or <code>null</code> if no such element is found
     * @throws ContextException when a error occurs
     */
    public IContextElement queryContextLastElement(
            final IEntity entity,
            final IScope scope)
            throws ContextException;

    /**
     * Provides synchronous access to the specified context value. Unlike the
     * {@link #queryContextLastElement(IEntity, IScope)} method, this one
     * returns a context value corresponding to the named value of the last
     * element as identified by the arguments. In practice, this method is
     * equal to calling {@link #queryContextLastElement(IEntity, IScope)} and
     * then calling {@link IContextElement#getContextData()} to get the
     * context data and finally calling
     * {@link IContextData#getContextValue(IScope)} with the specified context
     * value scope.
```

```
     *
     * @param entity describes the {@link IEntity} of the desired context
     * element
     * @param scope describes the {@link IScope} of the desired context
     * element
     * @param valueScope the {@link IScope} of the desired context value
     *
     * @return the {@link IContextValue} which corresponds to the named
     * context value of the requested entity/scope, or <code>null</code>
     * if no such element (or value) is found
     *
     * @throws ContextException when a error occurs
     */
    public IContextValue queryContextLastValue(
            final IEntity entity,
            final IScope scope,
            final IScope valueScope)
            throws ContextException;

    /**
     * Provides asynchronous access to the specified context data.
     *
     * @param entity describes the entity of the desired context data
     * @param scope describes the scope of the desired context data
     * @param listener typically a reference to the client which requests the
     * asynchronous notification for the given query
     * @throws ContextException when a error occurs
     */
    public void queryContext(
            final IEntity entity,
            final IScope scope,
            final IContextListener listener)
            throws ContextException;

    /**
     * Provides synchronous access to the context data.
     *
     * @param query a query describing the desired set of context data
     * @return a dataset containing the context which corresponds to the
     * issued query
     * @throws ContextException when a error occurs
     */
    public IContextDataset queryContext(final IContextQuery query)
            throws ContextException;

    /**
     * Provides asynchronous access to the context data. The provided context
     * listener is asynchronously contacted once the results of the given
```

```
     * {@link IContextQuery} are available.
     *
     * @param query a query describing the desired set of context data
     * @param listener typically a reference to the client which requests the
     * asynchronous notification for the given query
     * @throws ContextException when a error occurs
     */
    public void queryContext(
            final IContextQuery query,
            final IContextListener listener)
            throws ContextException;

    /**
     * Provides asynchronous access to the context data. By providing a the
     * {@link IEntity} and the {@link IScope} of the required data, the
     * provided {@link IContextListener} reference is asynchronously notified
     * of context changes in the specified entity-scope pair automatically.
     *
     * @param entity the {@link IEntity} of the requested data
     * @param scope the {@link IScope} of the requested data
     * @param listener typically a reference to the client which requests the
     * asynchronous notification for the entity-scope pair
     * @throws ContextException when a error occurs
     */
    public void addContextListener(
            final IEntity entity,
            final IScope scope,
            final IContextListener listener)
            throws ContextException;

    /**
     * Provides asynchronous access to the context data. This method is used
     * along with the
     * {@link #addContextListener(IEntity, IScope, IContextListener)} method
     * in order to allow unregistering of a context client from a particular
     * entity-scope pair.
     *
     * @param entity the {@link IEntity} of the requested data
     * @param scope the {@link IScope} of the requested data
     * @param listener typically a reference to the client which requests to
     * unregister from asynchronous notification for the given query
     * @throws ContextException when an error occurs
     */
    public void removeContextListener(
            final IEntity entity,
            final IScope scope,
            final IContextListener listener)
            throws ContextException;
```

```
/**
 * Provides the ability to the context clients to specify which context
 * types they require (i.e. because an application was launched, requiring
 * additional context types). The identification of a type is done in
 * terms of a doublet comprising of an {@link IEntity} and a
 * {@link IScope}.
 *
 * When registered as <i>needed</i>, a context type is monitored by the
 * context system be means of activating an appropriate context sensor (if
 * available). When multiple sensors are available providing the same
 * context type, then the system might decide to activate only one or more
 * of them, based on the resource availability and the quality of the
 * available sensors.
 *
 * Registering the same (Entity,Scope) doublet twice does not throw an
 * exception and does not result in any noticeable change.
 *
 * @param entity {@link Entity} the entity which is to be monitored
 * @param scope {@link Scope} the scope of the entity which must be
 * monitored
 * @param requestor the context client which makes the request
 * @throws NullPointerException if either of the arguments is null
 * @throws ContextException when a context-related exception occurs
 */
public void addNeededContextType(
        final IEntity entity,
        final IScope scope,
        final Object requestor)
        throws ContextException;

/**
 * Provides the ability to the context clients to specify which context
 * types they do not need anymore (i.e. because an application was shut
 * down). The identification of a type is done in terms of a doublet
 * comprising of an {@link IEntity} and a {@link IScope}.
 *
 * Unregistering a non existing (Entity,Scope) doublet does not throw an
 * exception and does not result in any noticeable change.
 *
 * @param entity {@link Entity} the entity which is to be monitored
 * @param scope {@link Scope} the scope of the entity which must be
 * monitored
 * @param requestor the context client which originally made the request
 * and now wishes to cancel the request
 * @throws NullPointerException if either of the arguments is null
 * @throws ContextException when a context-related exception occurs
 */
```

```
    public void removeNeededContextType(
            final IEntity entity,
            final IScope scope,
            final Object requestor)
            throws ContextException;
}
```

Listing A.1: The `IContextAccess` service

```
/*
 * The MUSIC project (Contract No. IST-035166) is an Integrated Project (IP)
 * within the 6th Framework Programme, Priority 2.5.5 (Software and Services).
 */

package org.istmusic.mw.context;

import org.istmusic.mw.context.events.IContextManagementListener;
import org.istmusic.mw.context.events.ContextChangedEvent;
import org.istmusic.mw.context.model.api.IEntity;
import org.istmusic.mw.context.model.api.IScope;
import java.util.Set;

/**
 * Provides management-level access concerning the internal state of the MUSIC
 * context system. For instance, it allows external components to be informed
 * of the currently needed and provided context types. Typical clients of this
 * interface are components implementing context distribution and context
 * visualization or simulation.
 */
public interface IContextManagement
{
    /**
     * Returns a {@link Set} of {@link EntityScopePair}s which correspond to
     * the currently required context types.
     *
     * @return a {@link Set} of {@link EntityScopePair}s which correspond to
     * the currently required context types.
     */
    public Set getCurrentRequiredContextTypes();

    /**
     * Registers for asynchronous notification of when a new required context
     * type (i.e. {@link EntityScopePair}) is added or removed.
     *
     * @param listener an implementation of the
     * {@link IContextManagementListener} interface to facilitated the
     * call-back pattern
     */
```

```
public void addRequiredContextTypesListener(
        final IContextManagementListener listener);

/**
 * Un-registers from asynchronous notification of when a new required
 * context type (i.e. {@link EntityScopePair}) is added or removed.
 *
 * @param listener an implementation of the
 * {@link IContextManagementListener} interface to facilitated the
 * call-back pattern
 */
public void removeRequiredContextTypesListener(
        final IContextManagementListener listener);

/**
 * Returns a {@link Set} of {@link EntityScopePair}s which correspond to
 * the currently provided context types.
 *
 * @return a {@link Set} of {@link EntityScopePair}s which correspond to
 * the currently required context types.
 */
public Set getCurrentProvidedContextTypes();

/**
 * Registers for asynchronous notification of when a new provided context
 * type (i.e. {@link EntityScopePair}) is added or removed.
 *
 * @param listener an implementation of the
 * {@link IContextManagementListener} interface to facilitated the
 * call-back pattern
 */
public void addProvidedContextTypesListener(
        final IContextManagementListener listener);

/**
 * Un-registers from asynchronous notification of when a new provided
 * context type (i.e. {@link EntityScopePair}) is added or removed.
 *
 * @param listener an implementation of the
 * {@link IContextManagementListener} interface to facilitated the
 * call-back pattern
 */
public void removeProvidedContextTypesListener(
        final IContextManagementListener listener);

/**
 * Returns a {@link Set} of {@link EntityScopePair}s which correspond to
 * the currently required context types from remote sources.
```

```
     *
     * @return a {@link Set} of {@link EntityScopePair}s which correspond to
     * the currently required context types.
     */
    public Set getCurrentRequiredRemoteContextTypes();

    /**
     * Registers for asynchronous notification of when a new required remote
     * context type (i.e. {@link EntityScopePair}) is added or removed.
     *
     * @param listener an implementation of the
     * {@link IContextManagementListener} interface to facilitated the
     * call-back pattern
     */
    public void addRequiredRemoteContextTypesListener(
            final IContextManagementListener listener);

    /**
     * Un-registers from asynchronous notification of when a new provided
     * context type (i.e. {@link EntityScopePair}) is added or removed.
     *
     * @param listener an implementation of the
     * {@link IContextManagementListener} interface to facilitated the
     * call-back pattern
     */
    public void removeRequiredRemoteContextTypesListener(
            final IContextManagementListener listener);

    /**
     * It checks if there is at least one active plug-in providing the
     * required context type. The context type is identified by the
     * specified {@link IEntity} and {@link IScope}.
     *
     * @param entity the entity of the requested context type
     * @param scope the scope of the requested context type
     *
     * @return true if there is at least one active plug-in offering the
     * specified context type, false otherwise
     */
    public boolean isActive(IEntity entity, IScope scope);

    /**
     * It checks if there is at least one resolved plug-in providing the
     * required context type. The context type is identified by the specified
     * {@link IEntity} and {@link IScope}.
     *
     * @param entity the entity of the requested context type
     * @param scope the scope of the requested context type
```

```
     *
     * @return true if there is at least one resolved plug-in offering the
     * specified context type, false otherwise
     */
    public boolean isResolved(IEntity entity, IScope scope);

    /**
     * It checks if there is at least one installed plug-in providing the
     * required context type. The context type is identified by the specified
     * {@link IEntity} and {@link IScope}.
     *
     * @param entity the entity of the requested context type
     * @param scope the scope of the requested context type
     *
     * @return true if there is at least one installed plug-in offering the
     * specified context type, false otherwise
     */
    public boolean isInstalled(IEntity entity, IScope scope);

    /**
     * Allows external entities (such as the "Distributed Context Manager") to
     * raise {@link ContextChangedEvent}s.
     *
     * @param event the event to be pushed in the context system.
     */
    public void contextChanged(final ContextChangedEvent event);

    /**
     * Similar to the {@link #contextChanged(ContextChangedEvent)} method,
         this
     * one also pushes the given event to the context system. However, unlike
     * the previous method, the given event is <b>not</b> intercepted. Rather,
     * it is forwarded to the context clients as needed.
     *
     * @param event the event to be pushed in the context system.
     */
    public void uninterceptedContextChanged(final ContextChangedEvent event);
}
```

Listing A.2: The `IContextManagement` service

```
/*
 * The MUSIC project (Contract No. IST-035166) is an Integrated Project (IP)
 * within the 6th Framework Programme, Priority 2.5.5 (Software and Services).
 */

package org.istmusic.mw.context;
```

```
import org.istmusic.mw.context.events.ContextChangedEvent;

/**
 * This interface specifies the functionality for context intercepionc which
 * is required to enable external components intercept the context events in
 * the middleware (for visualization purposes) and also simulate context
 * values (with the help of the {@link IContextManagement} service.
 */
public interface IContextSimulation
{
    /**
     * Automatically invoked by the context middleware for each context event
     * that is generated after the bundle is started.
     *
     * @param contextChangedEvent the event as originated from the context
     * middleware
     */
    public void interceptContextChangedEvent(
            final ContextChangedEvent contextChangedEvent);
}
```

Listing A.3: The `IContextSimulation` service

A.2 Required services

```
/*
 * The MUSIC project (Contract No. IST-035166) is an Integrated Project (IP)
 * within the 6th Framework Programme, Priority 2.5.5 (Software and Services).
 */

package org.istmusic.mw.context.cqp;

import org.istmusic.mw.context.cqp.queryapi.IContextQuery;
import org.istmusic.mw.context.exceptions.GenericQueryException;
import org.istmusic.mw.context.exceptions.QueryNotImplementedException;
import org.istmusic.mw.context.exceptions.QueryNotValidException;

import org.istmusic.mw.context.model.api.IContextElement;

/**
 * The IContextQueryService interface is exposed by the Context Query
    Processor
 * to the Context Manager as defined by the MUSIC reference architecture.
 */
public interface IContextQueryService
{
```

```
    /**
     * The processActiveAsynchronousQuery method processes an already active
     * asynchronous query.
     *
     * @param contextQuery the query to be processed
     * @param currentValue the last value of the context to be checked (if
     * applicable, e.g. ONCHANGE condition)
     * @return the result of the query as a response
     * @throws QueryNotValidException, QueryNotImplementedException,
     * GenericQueryException if an error occurs
     *
     */
    public QueryResponse processActiveAsynchronousQuery(
            final IContextQuery contextQuery,
            final IContextElement [] currentValue)
            throws QueryNotValidException, QueryNotImplementedException,
                GenericQueryException;

    /**
     * The processSynchronousQuery method processes a synchronous or an
     * asynchronous query.
     *
     * @param contextQuery the query to be processed
     * @return the result of the query as a response
     * @throws QueryNotValidException if the specified query is invalid
     * @throws QueryNotImplementedException if the specified query is not
     * implemented
     */
    public QueryResponse processQuery(final IContextQuery contextQuery)
            throws QueryNotValidException, QueryNotImplementedException;
}
```

Listing A.4: The `IContextQueryService` service

```
/*
 * The MUSIC project (Contract No. IST-035166) is an Integrated Project (IP)
 * within the 6th Framework Programme, Priority 2.5.5 (Software and Services).
 */

package org.istmusic.mw.context.model.api;

import org.istmusic.mw.context.exceptions.TransformationException;
import java.util.Set;
```

```
/**
 * Provides the functionality implemented by any provider of the context model
 * service. This functionality includes methods for introdpecting the model
 * and also for performing inter-representation transformations.
 */
public interface IContextModel
{
    /**
     * Returns a string representing the source of this model. For instance,
     * if the model was dynamically constructed based on an ontology, then the
     * name/id of the ontology is returned.
     *
     * @return a string representing the source of this model
     */
    public String getSource();

    /**
     * Returns a set containing all the {@link IEntity} entries defined in the
     * model.
     *
     * Introspection method.
     *
     * @return an instance of {@link java.util.Set}
     */
    public Set getAvailableEntities();

    /**
     * Returns a set containing all the {@link IScope} entries defined in the
     * model for the given entity.
     *
     * Introspection method.
     *
     * @param entity an instance of {@link IEntity}
     *
     * @return an instance of {@link Set}
     */
    public Set getAvailableScopes(final IEntity entity);

    /**
     * Returns the {@link IRepresentation}s defined in the model for the given
     * {@link IScope}.
     *
     * Introspection method.
     *
     * @param scope the specified
     * {@link IScope}
     * @return an instance of {@link Set}
     */
```

```
public Set getRepresentationsFor(final IScope scope);

/**
 * Returns the {@link IRelationship}s defined in the model.
 *
 * Introspection method.
 *
 * @return an instance of {@link Set}
 */
public Set getAvailableRelationships();

/**
 * Checks if the two entities specified have a directional relationship.
 * @param source the source entity
 * @param target the target entity
 * @param relationship the directional relationship to be checked
 * @return a boolean
 */
public boolean haveRelationship(
        final IEntity source,
        final IEntity target,
        final IRelationship relationship);

/**
 * Returns the directional relationships between the source and the target
 * entities.
 * @param source the source entity
 * @param target the target entity
 * @return an instance of {@link Set}
 */
public Set getRelationships(final IEntity source, final IEntity target);

/**
 * Returns a set, possibly <code>null</code>, containing all the
 * relationships where the specified entity is found to participate in
 * this model.
 *
 * @param entity the specified entity to be chacked
 * @return an instance of {@link Set}
 */
public Set getRelated(final IEntity entity);
```

```
    /**
     * Returns true if and only if the context specified by the given entity
     * and scope, can be transformed from the specified source representation
     * to the target one.
     *
     * @param entity the {@link IEntity} of the corresponding context
     * information
     * @param scope the {@link IScope} of the corresponding context
     * information
     * @param fromRepresentation the source representation
     * @param toRepresentation the target representation
     * @return true if and only if the context specified by the given entity
     * and scope, can be transformed from the specified source representation
     * to the target one.
     */
    public boolean supportsIROTransformation(
            final IEntity entity,
            final IScope scope,
            final IRepresentation fromRepresentation,
            final IRepresentation toRepresentation);

    /**
     * Tranforms the given context element from the source representation to
     * the target one.
     *
     * @param fromContextElement the source instance of
     * {@link IContextElement}
     * @param toRepresentation the target representation
     * @return a new instance of {@link IContextElement}, corresponding
     * to the requested representation.
     * @throws TransformationException when the transformation fails
     */
    public IContextElement performIROTransformation(
            final IContextElement fromContextElement,
            final IRepresentation toRepresentation)
        throws TransformationException;
}
```

Listing A.5: The `IContextModel` service

```
/*
 * The MUSIC project (Contract No. IST-035166) is an Integrated Project (IP)
 * within the 6th Framework Programme, Priority 2.5.5 (Software and Services).
 */

package org.istmusic.mw.context.cache;

import org.istmusic.mw.context.model.api.IContextElement;
```

```
import org.istmusic.mw.context.model.api.IEntity;
import org.istmusic.mw.context.model.api.IScope;

/**
 * This is the interface used by the context middleware and the Context Query
 * Processor to store and retrieve context information from the repository.
 */
public interface IContextRepository
{
    /**
     * Stores the specified context element in the repository.
     *
     * @param contextElement the context element to be stored
     */
    void storeContextElement(final IContextElement contextElement);

    /**
     * Retrieves all stored context elements from the repository.
     *
     * @param entity the {@link IEntity} of the context element to be returned
     * @param scope the {@link IScope} of the context element to be returned
     * @return all stored context elements from the repository as an array.
     */
    public IContextElement [] getAllContextElements(
            final IEntity entity,
            final IScope scope);

    /**
     * Retrieves all stored context elements from the repository, subject to
     * the specified conditions.
     *
     * @param entity the {@link IEntity} of the context element to be returned
     * @param scope the {@link IScope} of the context element to be returned
     * @param condition the {@link ICondition} based on which the context
     * values are checked
     * @return all stored context elements from the repository as an array.
     */
    public IContextElement [] getContextElement(
            final IEntity entity,
            final IScope scope,
            final ICondition condition
    );
```

```
    /**
     * Returns the last context element stored in the cache.
     *
     * @param entity the {@link IEntity} of the context element to be returned
     * @param scope the {@link IScope} of the context element to be returned
     *
     * @return the {@link IContextElement} which was stored in the cache last,
     * or <code>null</code> if no element was stored yet.
     */
    public IContextElement getLastContextElement(
            final IEntity entity,
            final IScope scope);

    /**
     * Triggers the context cache to perform Garbage Collection. During this
     * process, the cache is supposed to test if it has any obsolete values
     * which need be clared. The cache can possibly decide to pass this offer,
     * in which case it should return <i>false</i>.
     *
     * @return true if the cache has actually performed Garbage Collection
     * during this call, false otherwise
     */
    public boolean garbageCollect();
}
```

Listing A.6: The `IContextRepository` service

Appendix B

Summary of the classroom-based survey: questions and answers

This chapter presents the results of a questionnaire which was filled in by the students of the EPL-429 course (Spring 2009) at the University of Cyprus on Thursday, April 9th 2009 at 6:30pm. The students attended the course titled "Context-aware Pervasive Systems" which included lab assignments during which the students were guided to develop context-aware applications using the methodology and the middleware presented in this thesis.

A subset of the students have also used the MDD-based approach (see chapter 6) for creating context plug-ins that they had previously developed manually (i.e., without the modeling tools).

Parts A and B were answered by all 12 students. Parts C and D were only answered by the 4 students who received hands-on experience with the MDD-based approach. For sections A and B, the results of these 4 students are listed inside parentheses (i.e., "Yes $\rightarrow$ 10 (3)" means that 10 students in total answered 'Yes', 3 of whom also used the MDD approach). This separation is important as the feedback from these students was evidently more credible regarding the comparison of the manual development approach versus the MDD-based approach.

B.1 Participants information

- Number of participants: 12 (4)

- Year of study: 4th (10) and 3rd (2)
- Years of experience with programming: 3 to 4
- Years of experience with Java programming: 1 to 4 (evenly distributed)
- Experience with UML: Low to medium
- Experience with other modeling languages: No
- Experience with Model Driven Development: None to medium (some students used the Rational Rose tool as part of a software engineering class)

B.2 General questions

Question	**Yes**	**Partially**	**No**
A.1 - Was the concept of "context aware" applications clear to you?	11 (4)	1 (0)	0 (0)
A.2 - Are the benefits of using the "MUSIC Context Middleware" understandable to you?	9 (4)	3 (0)	0 (0)
A.3 - Are the following elements of the "MUSIC Context Middleware" clear to you?			
- OSGi framework	10 (4)	2 (0)	0 (0)
- Context middleware	8 (4)	4 (0)	0 (0)
- Context plug-ins	9 (4)	3 (0)	0 (0)
- Model-driven Development tools	6 (3)	4 (1)	2 (0)
A.4 - Overall, if your job was to develop context-aware applications, would you use the MUSIC middleware?	5 (2)	6 (2)	1 (0)

B.3 Manual implementation of context plug-ins

Preparation			
Question	**Answers**		
	High	**Medium**	**Low**
B.1 - Quality of lecture material	3 (2)	9 (2)	0 (0)
B.2 - Quality of related tutorials	9 (4)	3 (0)	0 (0)
B.3 - Quality of additional preparation material (e.g. books, web-sites, etc)	5 (1)	5 (2)	2 (1)
B.4 - Difficulty to understand the following concepts			
- Identify the relevant context types	0 (0)	5 (2)	7 (2)
- Design the architecture of the plug-in	2 (0)	7 (2)	3 (2)
- Define the MANIFEST file	0 (0)	7 (1)	5 (3)
- Define the service descriptor (XML) file	1 (0)	8 (2)	3 (2)
- Define the context model	4 (0)	7 (3)	1 (1)
- Define the code of the plug-in	0 (0)	8 (3)	4 (1)
- Package the plug-in in an OSGi bundle	3 (0)	5 (1)	4 (3)
B.5 - Most difficult concepts (concept + reason)	Context Model (2), Plug-in code (1)		
B.6 - Time for preparation (hours)	Avg=22.8 (5.5), Med=20 (10), Min=1 (1), Max=48 (10)		

Implementation			
Question	**Answers**		
B.7 - How much time was spent for implementation (hours)?	19.6 (16.0)		
B.8 - Have you succeeded to realize the plug-in?	**Yes**	**No**	
	12 (4)	0 (0)	
B.9 - How much time was required to understand the API (hours)?	3.5 (2.3)		
B.10 - How complex was the implementation?	**High**	**Medium**	**Low**
	1 (0)	9 (2)	2 (2)
B.11 - Which steps have been the most difficult?	Understand and use the MUSIC API, Define the plug-in code, Install & run plug-in, Manifest, Context model, Plug-in implementation, Package & deploy		
B.12 - How much help was required from the lecturer?	**Much**	**Some**	**Little**
	5 (2)	4 (0)	3 (2)
B.13 - For which steps was help required from the lecturer (step)?	Identify the relevant context types, Design the context model (3), Include needed libraries in plug-in, Design Manifest (2), Service descriptor, Package plug-in, Context model, Plug-in code		

B.4 Model-driven development of context plug-ins

Question	**Answers**		
Preparation			
	High	**Medium**	**Low**
C.1 - Quality of lecture material	6 (4)	6 (0)	0 (0)
C.2 - Quality of related tutorials	8 (4)	4 (0)	0 (0)
C.3 - Difficulty to understand the required concepts	1 (0)	7 (2)	3 (2)
C.4 - Additional preparation material (books, web, etc)	Web		
C.5 - Time for preparation	(3.75)		
Implementation			
C.6 - How many hours were spent for implementation?[1]	(1)		
C.7 - Have you succeeded to realize the Context Plug-in?	**Yes**	**Partially**	**No**
	(4)	(0)	(0)
C.8 - How much time was required to understand the modeling concepts (hours)?	(1.25)		
C.9 - How complex was the implementation?	**High**	**Medium**	**Low**
	(0)	(4)	(0)
C.10 - Which steps have been the most difficult?	Understand the concepts, Design the model (steep learning curve)		
C.11 - How much help was required from the lecturer?	**Much**	**Some**	**None**
	(4)	(0)	(0)
C.12 - For which steps was help required from the lecturer?	All		

[1]For the students who did try the MDD approach, the time includes just the design of the UML model and the transformation to source code (additional steps required to implement missing pieces of code were ignored)

B.5 Manual versus MDD-based implementation

D.1 - Which development approach do you think is easier and why?

- MDD because of high level abstractions
- MDD is easier if you are familiar with the tool (faster development)
- MDD because of many automations (MANIFEST, Service Descriptor, etc)
- Once you get familiar with EA, the MDD approach is much easier (otherwise manual implementation is preferred)

D.2 - Which development approach do you prefer and why?

- MDD for abstractions and platform independence
- I prefer the manual implementation because I am more familiar with it (but if I learned to use MDD then maybe I would change my mind)
- Manual implementation because the MDD-approach requires that you know the tools very well in order to use it
- MDD approach because it is easier (if you are familiar with the tool) and you save time

D.3 - Would you apply the MDD approach if you had to develop another plug-in?

- Yes (all 4 of them)

D.4 - How could the manual implementation task be simplified?

- Automatically generate MANIFEST and Service Descriptor
- Automatic generation of standardized functionality

D.5 - How could the MDD approach be improved?

- Automatic generation of the general plug-in schema
- Offer even more functionality

D.6 - Pros and Cons of manual implementation:

Pros:	**Cons:**
Freedom of coding	Architectural (Platform) dependent
Easy and fast for experience programmers	More time to develop
Simple Java code programming	More time / More bugs
You have better control (of the code)	More time consuming
You better understand what you are doing	

D.7 - Pros and Cons of the MDD approach:

Pros:	**Cons:**
Architectural independence	Slow
Easy for newbies	You do not always understand what is happening behind the model
Easier way to build your plug-in	
Less bugs	
Automation of certain functionalities	
Time saving	

Bibliography

[1] Ambient umbrella. http://www.ambientdevices.com/products/umbrella.html.

[2] APPLE iPhone. www.apple.com/iphone.

[3] Enkin. http://www.enkin.net.

[4] Google sky map. http://www.google.com/sky/skymap.html.

[5] JXTA (Juxtapose), Peer-to-peer protocol specification. https://jxta.dev.java.net.

[6] Mobile users in ubiquitous computing environments (MUSIC). http://www.ist-music.eu.

[7] Mobility and adaptation-enabling middleware (MADAM). http://www.ist-madam.org.

[8] Modelware. http://www.modelware-ist.org.

[9] Mofscript home page. http://www.eclipse.org/gmt/mofscript.

[10] NOKIA Symbian OS, Series 60 (S60). http://www.s60.com.

[11] Open Handset Alliance ANDROID Platform. http://www.android.com.

[12] Sparx systems: Enterprise architect. http://www.sparxsystems.com.

[13] Web Ontology Language (OWL), MOF-based metamodel, v. 1.1. http://www.webont.org/owl/1.1.

[14] A survey of system development process models. Survey CTG.MFA - 003, Center for Technology in Government, 1998.

[15] *Microsoft Visual C#(TM) .NET Language Reference*. Microsoft Press, Apr. 2002.

[16] *Measuring the Information Society: The ICT Development Index*. International Telecommunication Union, 2009.

[17] ABRAN, A., MOORE, J. W., BOURQUE, P., AND DUPUIS, R. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society, 2004.

[18] ALIA, HALLSTEINSEN, PASPALLIS, AND ELIASSEN. Managing distributed adaptation of mobile applications. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'07)* (Paphos, Cyprus, 2007), vol. 4531 of *LNCS*, Springer Verlag, pp. 104–118.

[19] ALIA, M., EIDE, V. S. W., PASPALLIS, N., ELIASSEN, F., HALLSTEINSEN, S. O., AND PAPADOPOULOS, G. A. A utility-based adaptivity model for mobile applications. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)* (Niagara Falls, Ontario, Canada, May 2007), IEEE Computer Society Press, pp. 556–563.

[20] ALIA, M., HORN, G., ELIASSEN, F., KHAN, M. U., FRICKE, R., AND REICHLE, R. A component-based planning framework for adaptive systems. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA'08)* (Montpellier, France, 2006), vol. 4277 of *LNCS*, Springer Verlag, pp. 1686–1704.

[21] ANGELES-PINA, C. *Distribution of context information using the Session Initiation Protocol (SIP)*. Master of science thesis, KTH Information and Communication Technology.

[22] ARNOLD, K., GOSLING, J., AND HOLMES, D. *The Java(TM) Programming Language*, 3rd ed. Prentice Hall PTR, June 2005.

[23] AYED, D., DELANOTE, D., AND BERBERS, Y. MDD approach for the development of context-aware applications. In *Proceedings of the 6th International and Interdisciplinary Conference on Modeling and Using Context (Context'07)* (Roskilde University, Denmark, 2007), vol. 4635 of *LNCS*, Springer Verlag, pp. 15–28.

[24] BAER, P. A., AND REICHLE, R. Communication and collaboration in heterogeneous teams of soccer robots. In *Robotic Soccer*. I-Tech Education and Publishing, Vienna, Austria, Dec. 2007, pp. 1–28.

[25] BANAVAR, G., BECK, J., GLUZBERG, E., MUNSON, J., SUSSMAN, J., AND ZUKOWSKI, D. Challenges: an application model for pervasive computing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking* (Boston, Massachusetts, United States, 2000), ACM, pp. 266–274.

[26] BELL, G., AND DOURISH, P. Yesterday's tomorrows: notes on ubiquitous computing's dominant vision. *Personal Ubiquitous Computing 11*, 2 (2007), 133–143.

[27] BRATSKAS, P., PASPALLIS, N., AND PAPADOPOULOS, G. A. An evaluation of the state of the art in context-aware architectures. In *Proceedings of the 16th International Conference on Information Systems Development (ISD'07)* (Galway, Ireland, 2007), Springer Verlag, pp. 1–12.

[28] BROOKS, R. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*, 1 (1986), 14–23.

[29] BROWN, P. J. The stick-e document: a framework for creating context-aware applications. *Electronic Publishing 8*, 2 & 3 (June 1996), 259–272.

[30] BROWN, P. J. Triggering information by context. *Personal and Ubiquitous Computing 2*, 1 (Mar. 1998), 18–27.

[31] BROWN, P. J., BOVEY, J. D., AND CHEN, X. Context-aware applications: from the laboratory to the marketplace. *IEEE Personal Communications 4*, 5 (1997), 58–64.

[32] CAPRA, L. CARISMA: context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering 29*, 10 (2003), 929–945.

[33] CAPRA, L., EMMERICH, W., AND MASCOLO, C. Reflective middleware solutions for context-aware applications. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION'01)* (Kyoto, Japan, 2001), vol. 2192 of *LNCS*, Springer Verlag, pp. 126–133.

[34] CAPRA, L., EMMERICH, W., AND MASCOLO, C. A micro-economic approach to conflict resolution in mobile computing. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (Charleston, South Carolina, USA, 2002), ACM, pp. 31–40.

[35] CERVANTES, H., AND HALL, R. S. Automating service dependency management in a service-oriented component model. In *Proceedings of the 6th Workshop on Component-Based Software Engineering (CBSE) in conjuction with the 25th Internation Conference on Software Engineering (ICSE)* (Portland, Oregon, USA, 2003).

[36] CERVANTES, H., AND HALL, R. S. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)* (Edinburg, Scotland, UK, 2004), IEEE Computer Society, pp. 614–623.

[37] CHARETTE, R. N. This car runs on code. *IEEE Spectrum* (Feb. 2009). (Available online at http://blogs.spectrum.ieee.org/articles/2009/02/this_car_runs_on_code_1.html).

[38] CHEN, G. *SOLAR: Building a context fusion network for pervasive computing*. PhD thesis, Dartmouth College, Aug. 2004.

[39] CHEN, G., LI, M., AND KOTZ, D. Design and implementation of a large-scale context fusion network. In *Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)* (Boston, Massachussets, USA, Aug. 2004), IEEE Computer Society, pp. 246–255.

[40] CHEVERST, K., DAVIES, N., MITCHELL, K., AND EFSTRATIOU, C. Using context as a crystal ball: rewards and pitfalls. *Personal Ubiquitous Computing 5*, 1 (2001), 8–11.

[41] CONAN, D., ROUVOY, R., AND SEINTURIER, L. Scalable processing of context information with COSMOS. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'07)* (Paphos, Cyprus, 2007), vol. 4531, Springer Verlag, pp. 210–224.

[42] DAVIES, N., WADE, S. P., FRIDAY, A., AND BLAIR, G. S. Limbo: A tuple space based platform for adaptive mobile applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)* (Toronto, Canada, 1997), pp. 291–302.

[43] DEVLIC, A. *Context-addressed communication dispatch*. Licentiate thesis, Royal Institute of Technology (KTH), Apr. 2009.

[44] DEY, A. K. *Providing architectural support for building context-aware applications*. PhD thesis, Georgia Institute of Technology, 2000.

[45] DEY, A. K. Understanding and using context. *Personal Ubiquitous Computing 5*, 1 (2001), 4–7.

[46] DEY, A. K., ABOWD, G. D., AND SALBER, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction 16*, 2 (2001), 97–166.

[47] DEY, A. K., ABOWD, G. D., AND WOOD, A. CyberDesk: a framework for providing self-integrating context-aware services. In *Proceedings of the 3rd international conference on Intelligent user interfaces* (San Francisco, California, USA, 1998), ACM, pp. 47–54.

[48] DIJKSTRA, E. W. On the role of scientific thought. In *Selected writings on computing: A Personal Perspective*. Springer-Verlag, New York, NY, USA, 1982, pp. 60–66.

[49] EFSTRATIOU, C., CHEVERST, K., DAVIES, N., AND FRIDAY, A. An architecture for the effective support of adaptive context-aware applications. In *Proceedings of the 2nd International Conference on Mobile Data Management (MDM'01)* (Hong Kong, 2001), vol. 1987, Springer-Verlag, pp. 15–26.

[50] FLOCH, J., HALLSTEINSEN, S., STAV, E., ELIASSEN, F., LUND, K., AND GJORVEN, E. Using architecture models for runtime adaptability. *IEEE Software 23*, 2 (2006), 62–70.

[51] FREEMAN, E., HUPFER, S., AND ARNOLD, K. *JavaSpaces(TM) principles, patterns, and practice*. Prentice Hall PTR, June 1999.

[52] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, Nov. 1994.

[53] GEIHS, K., BAER, P., REICHLE, R., AND WOLLENHAUPT, J. Ontology-based automatic model transformations. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM '08)* (Cape Town, South Africa, 2008), IEEE Computer Society, pp. 387–391.

[54] GEIHS, K., BARONE, P., ELIASSEN, F., FLOCH, J., FRICKE, R., GJORVEN, E., HALLSTEINSEN, S., HORN, G., KHAN, M. U., MAMELLI, A., PAPADOPOULOS, G. A., PASPALLIS, N., REICHLE, R., AND STAV, E. A comprehensive solution for application-level adaptation. *Software: Practice and Experience 39*, 4 (2009), 385–422.

[55] GEIHS, K., REICHLE, R., WAGNER, M., AND KHAN, M. Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*, vol. 5525 of *LNCS*. Springer Verlag, 2009, pp. 146–163.

[56] GELERNTER, D. Generative communication in linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 80–112.

[57] GELERNTER, D., AND CARRIERO, N. Coordination languages and their significance. *Communications of the ACM 35*, 2 (1992), 97–107.

[58] GEORGALAS, N., OU, S., AZMOODEH, M., AND YANG, K. Towards a model-driven approach for ontology-based context-aware application development: a case study. In *Proceedings of the 4th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, 2007 (MOMPES '07)* (Braga, Portugal, Mar. 2007), IEEE Computer Society, pp. 21–32.

[59] GIBBS, W. W. Software's chronic crisis. *Scientific American* (Sept. 1994), 86.

[60] GOKHALE, A., SCHMIDT, D. C., NATARAJAN, B., AND WANG, N. Applying model-integrated computing to component middleware and enterprise applications. *Communications of the ACM 45*, 10 (2002), 65–70.

[61] GREENFIELD, A. *Everyware: The dawning age of ubiquitous computing*, 1st ed. New Riders Publishing, Mar. 2006.

[62] GREER, D. The art of separation of concerns, 2009. (Available online at http://ctrl-shift-b.com/2008/01/art-of-separation-of-concerns.html).

[63] HALLSTEINSEN, S., FLOCH, J., AND STAV, E. A middleware centric approach to building self-adapting systems. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM'05)* (Lisbon, Portugal, 2005), vol. 3437 of *LNCS*, Springer Verlag, pp. 107–122.

[64] HALPIN, T. *Information modeling and relational databases: from conceptual analysis to logical design*, 1st ed. Morgan Kaufmann, Apr. 2001.

[65] HEINEMAN, G. T., AND COUNCILL, W. T. *Component-based software engineering: putting the pieces together*. Addison-Wesley Professional, June 2001.

[66] HENRICKSEN, K. *A framework for context-aware pervasive computing applications*. PhD thesis, The University of Queensland, Sept. 2003.

[67] HENRICKSEN, K., AND INDULSKA, J. A software engineering framework for context-aware pervasive computing. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications (PerCom'04)* (Orlando, Florida, USA, Mar. 2004), IEEE Computer Society, pp. 77–86.

[68] HENRICKSEN, K., AND INDULSKA, J. Developing context-aware pervasive computing applications: models and approach. *Pervasive and Mobile Computing 2*, 1 (Feb. 2006), 37–64.

[69] HENRICKSEN, K., INDULSKA, J., MCFADDEN, T., AND BALASUBRAMANIAM, S. Middleware for distributed context-aware systems. In *Proceedings of the 7th International Conference On Distributed Objects And Applications (DOA'05)* (Agia Napa, Cyprus, 2005), vol. 3760 of *LNCS*, Springer-Verlag, pp. 846–863.

[70] HENRICKSEN, K., INDULSKA, J., AND RAKOTONIRAINY, A. Using context and preferences to implement self-adapting pervasive computing applications. *Software: Practice and Experience 36*, 11-12 (2006), 1307–1330.

[71] HIRSCHFELD, R., COSTANZA, P., AND NIERSTRASZ, O. Context-oriented programming. *Journal of Object Technology 7*, 3 (Apr. 2008), 125–151.

[72] HONG, J. I. *An architecture for privacy-sensitive ubiquitous computing*. PhD thesis, University of California, Berkeley, 2005.

[73] HONG, J. I., AND LANDAY, J. A. An infrastructure approach to context-aware computing. *Human-Computer Interaction 16*, 2 (2001), 287–303.

[74] HORN, P. Autonomic computing: IBM's perspective on the state of information technology, 2001.

[75] HU, X., DING, Y., PASPALLIS, N., BRATSKAS, P., PAPADOPOULOS, G. A., VANROMPAY, Y., PINHEIRO, M. K., AND BERBERS, Y. A hybrid peer-to-peer solution for context distribution in mobile and ubiquitous environments. In *Information Systems Development* (Paphos, Cyprus, 2009), Springer, pp. 501–510.

[76] HU, X., DING, Y., PASPALLIS, N., PAPADOPOULOS, G. A., BRATSKAS, P., BARONE, P., MAMELLI, A., VANROMPAY, Y., AND BERBERS, Y. A peer-to-peer based infrastructure for context distribution in mobile and ubiquitous environments. In *Proceedings of the 3rd International Workshop on Context-Aware Mobile Services (CAMS) in conjunction with the On The Move to Meaningful Internet Systems (OTM 2007)* (Vilamoura, Algarve, Portugal, 2007), vol. 4805 of *LNCS*, Springer Verlag, pp. 236–239.

[77] JACQUET, C., BOURDA, Y., AND BELLIK, Y. A component-based platform for accessing context in ubiquitous computing applications. *Journal of Ubiquitous Computing and Intelligence 1*, 2 (2007), 163–172.

[78] JUDD, G., AND STEENKISTE, P. Providing contextual information to pervasive computing applications. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications* (Dallas-Fort Worth, Texas, USA, Mar. 2003), IEEE Computer Society, p. 133.

[79] JULIEN, C., AND ROMAN, G. EgoSpaces: facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering 32*, 5 (2006), 281–298.

[80] KAKOUSIS, K., PASPALLIS, N., AND PAPADOPOULOS, G. A. Optimizing the utility function-based self-adaptive behavior of context-aware systems using user feedback. In *Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications (DOA'08)* (Monterrey, Mexico, 2008), vol. 5331 of *LNCS*, Springer Verlag, pp. 657–674.

[81] KALMAN, R. A new approach to linear filtering and prediction problems. *Transactions of the ASME Journal of Basic Engineering*, 82 (Series D) (1960), 35–45.

[82] KICZALES, G., BOBROW, D. G., AND DES RIVIRES, J. *The art of the metaobject protocol*. MIT Press, July 1991.

[83] KICZALES, G., AND MEZINI, M. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)* (Glasgow, Scotland, UK, 2005), vol. 3586 of *LNCS*, Springer Verlag, pp. 195–213.

[84] KIRSCH-PINHEIRO, M., VANROMPAY, Y., VICTOR, K., BERBERS, Y., VALLA, M., FR, C., MAMELLI, A., BARONE, P., HU, X., DEVLIC, A., AND PANAGIOTOU, G. Context grouping mechanism for context distribution in ubiquitous environments. In *Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications (DOA'08)* (Monterrey, Mexico, 2008), vol. 5331 of *LNCS*, Springer Verlag, pp. 571–588.

[85] KUMAR, S., COHEN, P., AND LEVESQUE, H. The adaptive agent architecture: achieving fault-tolerance using persistent broker teams. In *Proceedings of the 4th International Conference on MultiAgent Systems* (Boston, Massachusetts, USA, 2000), IEEE Computer Society, pp. 159–166.

[86] LONG, S., KOOPER, R., ABOWD, G. D., AND ATKESON, C. G. Rapid prototyping of mobile context-aware applications: the cyberguide case study. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking* (Rye, New York, USA, 1996), ACM, pp. 97–107.

[87] MAES, P. Concepts and experiments in computational reflection. *ACM SIGPLAN Notices 22*, 12 (1987), 147–155.

[88] MASCOLO, C., CAPRA, L., AND EMMERICH, W. Mobile computing middleware. In *Advanced Lectures on Networking*, vol. 2497 of *LNCS*. Springer Verlag, 2002, pp. 506–510.

[89] MCFADDEN, T., HENRICKSEN, K., AND INDULSKA, J. Automating context-aware application development. In *Proceedings of the 1st International Workshop on Advanced Context Modelling, Reasoning and Management in conjuction with UbiComp (UbiComp'04)* (Nottingham, England, UK, 2004), pp. 90–95.

[90] MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., AND CHENG, B. H. C. Composing adaptive software. *IEEE Computer 37*, 7 (2004), 56–64.

[91] MCKINLEY, P. K., SADJADI, S. M., KASTEN, E. P., AND CHENG, B. H. C. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 2004.

[92] MIKALSEN, M., FLOCH, J., PASPALLIS, N., PAPADOPOULOS, G. A., AND RUIZ, P. A. Putting context in context: the role and design of context management in a mobility and adaptation enabling middleware. In *Proceedings of the International Workshop on Managing Context Information and Semantics in Mobile Environments (MCISME'06) in conjunction with the 7th International Conference on Mobile Data Management (MDM'06)* (Nara, Japan, 2006), IEEE Computer Society, p. 76.

[93] MIKALSEN, M., PASPALLIS, N., FLOCH, J., STAV, E., PAPADOPOULOS, G. A., AND CHIMARIS, A. Distributed context management in a mobility and adaptation enabling middleware (MADAM). In *Proceedings of the 21st Annual ACM Symposium of Applied Computing, Track of Dependable and Adaptive Systems (SAC'06)* (Dijon, France, 2006), ACM, pp. 733–734.

[94] MINAR, N., GRAY, M., ROUP, O., KRIKORIAN, R., AND MAES, P. Hive: distributed agents for networking things. In *Proceedings of the 1st International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents* (Palm Springs, California, USA, 1999), IEE, pp. 118–129.

[95] MONSON-HAEFEL, R., BURKE, B., AND LABOUREY, S. *Enterprise JavaBeans*. O'Reilly, 2004.

[96] NAUR, P., AND RANDELL, B. *Software engineering: report of a conference sponsored by the NATO science committee*. Scientific Affairs Division, NATO, Garmisch, Germany, Oct. 1969.

[97] NEWBERGER, A., AND DEY, A. K. Designer support for context monitoring and control. Tech. Rep. IRB-TR-03-016, Intel Research Berkeley, 2003.

[98] NOBLE, B. System support for mobile, adaptive applications. *IEEE Personal Communications 7*, 1 (2000), 44–49.

[99] NORDBERG, M. E. Aspect-oriented dependency inversion. In *Proceedings of the Workshop on Advanced Separation of Concerns in Object- Oriented Systems in conjuction with Object Oriented Programming, Systems and Languages (OOPSLA'2001)* (Tampa Bay, Florida, USA, Oct. 2001).

[100] PADOVITZ, A., LOKE, S., AND ZASLAVSKY, A. Towards a theory of context spaces. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops* (Orlando, Florida, USA, 2004), IEEE Computer Society, pp. 38–42.

[101] PARKIN, M., MEYER, T., RUSH, M., AND BADE, R. *Foundations of microeconomics*, 2nd ed. Addison Wesley, Mar. 2003.

[102] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (1972), 1053–1058.

[103] PASCOE, J. The stick-e note architecture: extending the interface beyond the user. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces* (Orlando, Florida, USA, 1997), ACM, pp. 261–264.

[104] PASPALLIS, N., CHIMARIS, A., AND PAPADOPOULOS, G. A. Experiences from developing a distributed context management system for enabling adaptivity. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'07)* (Paphos, Cyprus, 2007), vol. 4531 of *LNCS*, Springer Verlag, pp. 225–238.

[105] PASPALLIS, N., ELIASSEN, F., HALLSTEINSEN, S., AND PAPADOPOULOS, G. A. Developing self-adaptive mobile applications and services with separation-of-concerns. In *At Your Service: Service-Oriented Computing from an EU Perspective*, E. D. Nitto, A. Sassen, and A. Zwegers, Eds. MIT Press, 2009, pp. 129–158.

[106] PASPALLIS, N., KAKOUSIS, K., AND PAPADOPOULOS, G. A. A multi-dimensional model enabling autonomic reasoning for context-aware pervasive applications. In *Proceedings of the Workshop for Human Control of Ubiquitous Systems (HUCUBIS'08) in conjunction with the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous'08)* (Trinity College Dublin, Ireland, July 2008), ACM Press.

[107] PASPALLIS, N., AND PAPADOPOULOS, G. A. An approach for developing adaptive, mobile applications with separation of concerns. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)* (Chicago, USA, 2006), vol. 1, IEEE Computer Society Press, pp. 299–306.

[108] PASPALLIS, N., AND PAPADOPOULOS, G. A. Distributed adaptation reasoning for a mobility and adaptation enabling middleware. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA'06)* (Montpellier, France, 2006), vol. 4277 of *LNCS*, Springer Verlag, pp. 17–18.

[109] PASPALLIS, N., AND PAPADOPOULOS, G. A. An optimization of context sharing for self-adaptive mobile applications. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'08)* (Agia Napa, Cyprus, 2008), vol. 5022 of *LNCS*, Springer Verlag, pp. 157–168.

[110] PASPALLIS, N., ROUVOY, R., BARONE, P., PAPADOPOULOS, G. A., ELIASSEN, F., AND MAMELLI, A. A pluggable and reconfigurable architecture for a context-aware enabling middleware system. In *Proceedings of the 10th International Symposium on Distributed Objects, Middleware, and Applications (DOA'08)* (Monterrey, Mexico, 2008), vol. 5331 of *LNCS*, Springer Verlag, pp. 553–570.

[111] PAYMANS, T. F., LINDENBERG, J., AND NEERINCX, M. Usability trade-offs for adaptive user interfaces: ease of use and learnability. In *Proceedings of the 9th International Conference on Intelligent User Interfaces* (Funchal, Madeira, Portugal, 2004), ACM, pp. 301–303.

[112] PERICH, F., JOSHI, A., YESHA, Y., AND FININ, T. Collaborative joins in a pervasive computing environment. *The VLDB Journal 14*, 2 (2005), 182–196.

[113] READE, C. *Elements of functional programming*. Addison Wesley Publishing Company, Boston, MA, USA, Apr. 1989.

[114] REICHLE, R., WAGNER, M., KHAN, M., GEIHS, K., LORENZO, J., VALLA, M., FRA, C., PASPALLIS, N., AND PAPADOPOULOS, G. A. A comprehensive context modeling framework for pervasive computing systems. In *Proceedings of the 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08)* (Oslo, Norway, 2008), vol. 5053 of *LNCS*, Springer Verlag, pp. 281–295.

[115] REICHLE, R., WAGNER, M., KHAN, M. U., GEIHS, K., VALLA, M., FRA, C., PASPALLIS, N., AND PAPA, G. A. A context query language for pervasive computing environments. In *Proceedings of the 5th IEEE Workshop on Context Modeling and Reasoning (CoMoRea'08) in conjunction with the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom'08)* (Hong Kong, Mar. 2008), IEEE Computer Society, pp. 434–440.

[116] ROUVOY, R., BARONE, P., DING, Y., ELIASSEN, F., HALLSTEINSEN, S., LORENZO, J., MAMELLI, A., AND SCHOLZ, U. MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*. 2009, pp. 164–182.

[117] ROUVOY, R., CONAN, D., AND SEINTURIER, L. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online 9*, 6 (2008), 1.

[118] ROUVOY, R., ELIASSEN, F., FLOCH, J., HALLSTEINSEN, S., AND STAV, E. Composing components and services using a planning-based adaptation middleware. In *Proceeding of the 7th International Symposium on Software Composition (SC)* (Budapest, Hungary, 2008), vol. 4954 of *LNCS*, Springer, pp. 52–67.

[119] RUBIN, A. The future of mobile, Sept. 2008. (Available online at http://googleblog.blogspot.com/2008/09 /future-of-mobile.html).

[120] SALBER, D., DEY, A. K., AND ABOWD, G. D. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems* (Pittsburgh, Pennsylvania, USA, 1999), ACM, pp. 434–441.

[121] SATYANARAYANAN, M. Fundamental challenges in mobile computing. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)* (Philadelphia, Pennsylvania, USA, 1996), ACM, pp. 1–7.

[122] SATYANARAYANAN, M. Pervasive computing: vision and challenges. *IEEE Personal Communications [see also IEEE Wireless Communications] 8*, 4 (2001), 10–17.

[123] SCHANTZ, R. E., AND SCHMIDT, D. C. Middleware for distributed systems - evolving the common structure for network-centric applications. *Encyclopedia of Software Engineering* (2001).

[124] SCHILIT, B. N. *A context-aware system architecture for mobile distributed computing*. PhD thesis, Columbia University, May 1995.

[125] SCHILIT, B. N., ADAMS, N. I., AND WANT, R. Context-aware computing applications. In *Proceedings of the 1st Workshop on Mobile Computing Systems and Applications (WMCSA'94)* (Santa Cruz, CA, Dec. 1994), IEEE Computer Society, pp. 85–90.

[126] SCHMIDT, D. C. Middleware for real-time and embedded systems. *Communications of the ACM 45*, 6 (2002), 43–48.

[127] SCHMIDT, D. C. Model driven engineering. *IEEE Computer 39*, 2 (Feb. 2006), 25–31.

[128] SOMMERVILLE, I. *Software engineering*, 7th ed. Addison Wesley, June 2004.

[129] SOUSA, J. P., AND GARLAN, D. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance* (Montreal, Quebec, Canada, 2002), Kluwer, B.V, pp. 29–43.

[130] STRANG, T., LINNHOFF-POPIEN, C., AND FRANK, K. CoOL: a context ontology language to enable contextual interoperability. In *Proceedings of the 4th IFIP Distributed Applications and Interoperable Systems (DAIS'03)* (Paris, France, 2003), vol. 2893 of *LNCS*, Springer Verlag, pp. 236–247.

[131] SZYPERSKI, C. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, Dec. 1997.

[132] TAVARES, A. L., AND VALENTE, M. T. A gentle introduction to OSGi. *SIGSOFT Software Engineering Notes 33*, 5 (2008), 1–5.

[133] TENNENHOUSE, D. Proactive computing. *Communications of the ACM 43*, 5 (2000), 43–50.

[134] WAGELAAR, D., AND JONCKERS, V. Explicit platform models for MDA. In *Proceedings of the ACM/IEEE 8th Model Driven Engineering Languages and Systems (MODELS'05)* (Genova, Italy, 2005), vol. 3713 of *LNCS*, Springer Verlag, pp. 367–381.

[135] WALSH, W. E., TESAURO, G., KEPHART, J. O., AND DAS, R. Utility functions in autonomic systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)* (New York, NY, USA, May 2004), IEEE Computer Society Press, pp. 70–77.

[136] WANG, X. H., ZHANG, D. Q., GU, T., AND PUNG, H. K. Ontology based context modeling and reasoning using OWL. In *Proceedings of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops* (Orlando, Florida, USA, 2004), IEEE Computer Society, p. 18.

[137] WANT, R., HOPPER, A., FALCO, V., AND GIBBONS, J. The active badge location system. *ACM Transactions on Information Systems 10*, 1 (1992), 91–102.

[138] WANT, R., SCHILIT, B., ADAMS, N., GOLD, R., PETERSEN, K., GOLDBERG, D., ELLIS, J., AND WEISER, M. The parctab ubiquitous computing experiment. In *Mobile Computing*, vol. 353 of *The Springer International Series in Engineering and Computer Science*. Springer, 1996, pp. 45–101.

[139] WEISER, M. The computer for the 21st century. *Scientific American* (Feb. 1991).

[140] WEISER, M. Hot topics: ubiquitous computing. *IEEE Computer 26*, 10 (Oct. 1993), 71–72.

[141] WOOD, A., DEY, A., AND ABOWD, G. D. CyberDesk: automated integration of desktop and network services. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA, 1997), ACM, pp. 552–553.

[142] WRIGHT, A. Get smart. *Communications of the ACM 52*, 1 (2009), 15–16.

[143] WRIGHT, A. Making sense of sensors. *Communications of the ACM 52*, 2 (2009), 14–15.

[144] WYCKOFF, P., MCLAUGHRY, S. W., LEHMAN, T. J., AND FORD, D. A. T spaces. *IBM Systems Journal 37*, 3 (1998), 454–474.

[145] XIAO, T. G., WANG, X. H., PUNG, H. K., AND ZHANG, D. Q. An ontology-based context model in intelligent environments. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'04)* (San Diego, California, USA, 2004), pp. 270–275.

[146] YAU, S. S., KARIM, F., WANG, Y., WANG, B., AND GUPTA, S. K. S. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing 1*, 3 (2002), 33–40.