# Building Context-Awareness Models for Mobile Applications

Chantal Taconet[1], Zakia Kazi-Aoul[2]
[1]Institut Télécom; Télécom SudParis; CNRS UMR SAMOVAR
9 Rue Charles Fourier, 91011, Évry Cedex, FRANCE
Chantal.Taconet@it-sudparis.eu

[2]ISEP; 28 Rue Notre Dame des Champs, 75006 Paris Cedex, FRANCE
zakia.kazi@isep.fr

*Journal of Digital Information Management*

***ABSTRACT:*** *The design process followed to produce traditional applications needs to be enhanced to cope with new context-aware ubiquitous application requirements. With the popularity of ubiquitous computing, context-aware applications become clearly necessary. This new kind of applications allows mobile users to universally access services in respect to any context including his computing environment. Challenges for the design of such applications are to easily define context collection requirements, context analysis and adaptations of the applications due to changes in its environment. To face these issues, we propose, in this article, a generic and extensible way to model context-awareness of any application using the model-driven engineering (MDE) approach. For this purpose, we add a context-awareness aspect to application model views. We illustrate our solution by modeling a context-aware e-commerce application. The addition of a context-awareness aspect, should ease the definition of mobile applications. Furthermore, context-awareness models open the way to automate context-awareness code production.*

**Categories and Subject Descriptors**
**I.3.4 [Graphics Utilities];** Meta files*:* **I.3.5 [Computational Geometry and Object Modeling]: E.2 Data Storage Representations**: C.1.3 Other Architecture Styles [Cellular architecture (e.g., mobile)*]*

**General Terms:** Context-Awareness Models, Mobile Applications, Ubiquitous computing, E Commerce

**Keywords:** Context-awareness, Model driven engineering, Ubiquitous computing

**Received:** 10 July 2009; Revised 17 August 2009; Accepted 28 August 2009

## 1. Introduction

With ubiquitous computing, users access their applications in a wide variety of environments. To cope with various and dynamic execution environments, context-aware applications emerge. One of the first use of the term "context-aware applications" appeared in 1994 (Schilit et al., 1994). According to Dey's definition (Dey, 2000), *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*. More generally, the goal of this kind of applications is to ensure universal access to applications in any context and with an application behavior which best suits the user's computing environment. Considering a mobile application, examples of unit of context are the user's terminal battery level and his terminal network connectivity bandwidth.

In the past years, many services have been designed to manage the context, among them Context- Toolkit (Dey, 2000), CONON (Wang *et al*., 2004), and COSMOS (Conan *et al*., 2007). The main goals of those services are context collection (i.e. how context values are dynamically collected from raw sensors or by derivation process) and context analysis (i.e. how to detect context modifications relevant for context-aware applications). One issue for context-aware applications is to easily make use of these various services at a low development cost and with easy reconfiguration enablers.

For context-aware application designers, the main difficulties come from the multiple kinds of context collectors and the asynchronism between the context management task and the application task (i.e. the detection of adaptation situations should not be included in the application logic). Furthermore, each context-aware application has its own behavior to react to context modifications. Finally, it would be interesting if this behavior could evolve easily during the life of the context-aware application (e.g. addition of an adaptation not foreseen at the creation of the application). For all these reasons, we argue that a promising solution for designing context-aware applications is to define their context-awareness following a Model Driven Engineering (MDE) approach.

MDE provides tools and grammars allowing the construction of context models which may be used to model context-aware systems. Context-awareness implementation can be generated automatically by transforming models to particular target platforms. This eases context-awareness evolution: any change in the context-awareness can be easily made at the model level and propagated automatically to the implementation. Furthermore, the time and effort of context-aware development can be reduced. Finally, it enhances portability and flexibility: the context-awareness implementation may easily make use of new context management services.

We propose, in this article, a generic and extensible process to design context-awareness in an application. Application context-awareness designers may select which elements have to be observed in the environment. For each of them, they may define an observation contract. Finally, they may also define variations in the application. All these definitions are made at a model level. MDE technologies help then to automate code generation concerning: (1) interactions with context management services and (2) runtime variations in the applications.

The outline of the paper is as follows. In Section 2, we explain our motivations through an illustrative e-commerce scenario, then, we present the whole design process overview and we give a general view of our proposed meta-models. Section 3 introduces the context specification and design phase conducted by context designers, whereas Section 4 details the context-awareness design phase conducted by the application context-awareness designers. In Section 5, we present our model production tools.Then, we compare our contribution

with regard to related work on context-awareness modelling in Section 6. Finally, we conclude and present some perspectives of our work in Section 7.

## 2. Motivations and illustrating scenario

We begin this section by presenting an illustrative scenario with a mobile context-aware e-commerce application (Section 2.1). Then, we introduce the terminology used in the sequel of the paper and we bring out our objectives in Section 2.3. We give a short overview of the context-aware application design process in Section 2.4, Finally, we draw a general picture of all the proposed meta-models in Section 2.5.

### 2.1. E-commerce illustrating scenario

The illustrative scenario is represented in Figure 1. Suzanne is a client of a famous e-commerce merchant where she often makes purchases of all kinds. Suzanne's client profile is used to offer Suzanne a customized service. Suzanne decides to go by train to visit her friend who lives near the beach. Once inside the train, she turns on her cell phone and uses its Internet connexion (e.g. 3G) to connect to the e-commerce server. When connected, Suzanne receives "offers" on: (i) hiking shoes because Suzanne's hobby is hiking, (ii) DVDs because today is Suzanne's best friend birthday, Betty, and Betty's hobby is cinema, and (iii) pullovers because the outside temperature measured by the weather station near the current mobile cell is 5 degrees Celsius.

Just when Suzanne decided to look in detail at the products using the application product description, the battery level reaches its low level. In order to save some battery power for her incoming calls, Suzanne has configured her profile to download videos only if the battery is not too low. Thus, the application switches to a poor mode omitting videos and animations. Images are allowed but must suit Suzanne's terminal screen size. Once Suzanne's terminal battery is plugged, the application product description returns back to the normal mode.

This scenario shows that the e-commerce ubiquitous application needs to be context-aware in order to cope with different user profiles and preferences, different terminal capabilities, and different elements of the environment in a distributed setting.

### 2.2. Terminology

For ubiquitous applications, such as the e-commerce application presented above, the context-awareness designer has to select, what to observe in the environment, what situations to detect and what adaptations to trigger in the application. For this purpose, we have chosen simple concepts that context-awareness designers should manipulate. We introduce in this section those concepts and the associated terminology.

**Entity:** En entity is an element representing a physical or logical phenomenon (person, concept, etc.) which can be treated as an independent unit or a member of a particular category, and to which "observables" may be associated. For example, Suzanne's terminal is an entity.

**Observable:** An observable is an abstraction which defines something to watch over (observe). For example, Suzanne's terminal battery level is an observable. Each observable may be observed by one or more software collectors (e.g., depending on the system, the collector to obtain the battery level will be different, several sensors may be used to obtain a temperature). Some observables may be computed from other observables, they are called *interpreted observables*.

**Observation:** An observation represents the state of an observable at a given time. It is obtained from a context-manager named collector in the sequel of the paper.

**Adaptation situation:** Some observables allow to track down a change of state in the space of the information of context. This change of state may require a reaction in the system. An observable which provides adaptation situations is, for example, Suzanne's terminal battery state which value could be "LowBattery" or "NormalBattery".

**Adaptation:** A modification in the system such as a structure modification (assembly change, component change) or a behavior change (change in a sequence of operations) is called an *adaptation*.

**Observation, notification and adaptation contracts:** An ob-

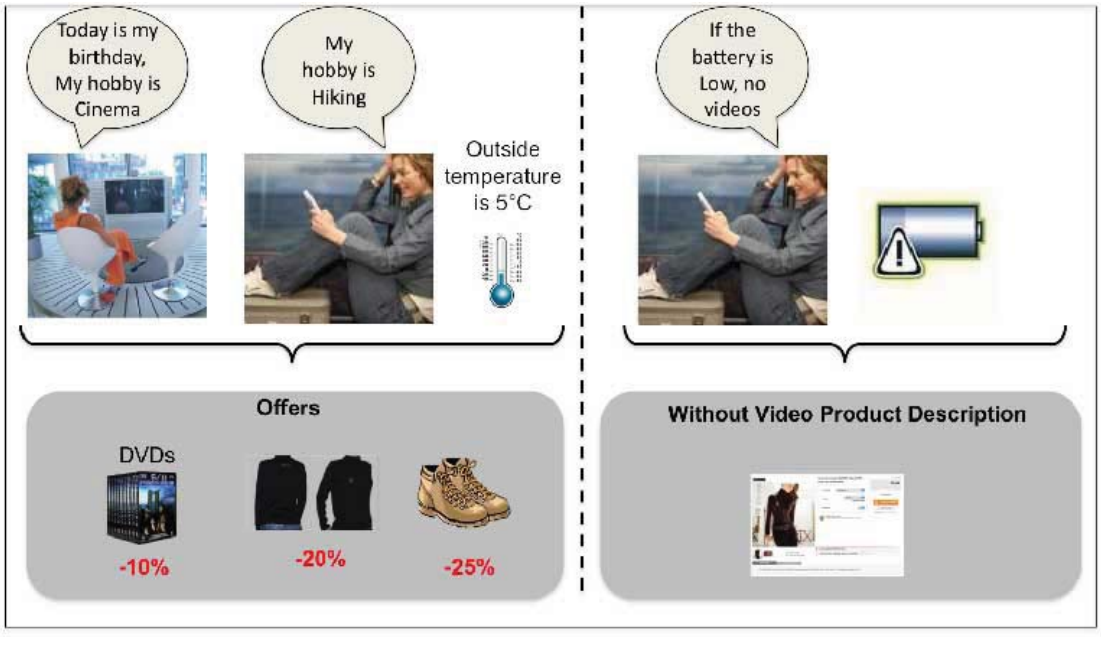


Figure 1. The illustrative scenario

servation of the environment may be defined by contract. The contract defines for example (i) if the application observes or needs to be notified, (ii) in case of notification which modifications trigger notifications, (iii) which adaptations are necessary in the context-aware system.

### 2.3. Motivations and objectives

Our work is driven following three objectives which the previous scenario induce. The first objective is the possibility to choose between different sources (several software collectors for a given observable) in order to collect observations for a given observable. Thus, the location and even the implementation of collectors is unknown during the modelling process. It provides more flexibility during the execution. As an example, the temperature value could be obtained either by the weather station near the current mobile or by a centralised weather station.

The second objective concerns distributed monitoring which is necessary when an application needs to monitor two or more users contexts to make adaptation decisions. For example, the "Offers" plugin needs to know Suzanne's hobbies and Betty's birthday as well as Betty's hobbies. We consider in this case that Suzanne's (or Betty's) context includes its hobbies and its friends birthdays.

Finally, our solution must deal with the evolution of the context-awareness model during runtime. Indeed, the application may require adaptation reconfiguration actions to face context changes or the availability of new context collectors.

### 2.4. Context-aware application design process

Figure 2 depicts the CA3M context-awareness design process with, from left to right, the stakeholders, the activities, and the resulting artefacts.

The figure illustrates two main activities: context specification and design and application design. The context specification and design comprises the design of collectors and the specification of contexts. It produces two kinds of artefacts: implementations and models. This modelling task is presented in Sections 3.1 and 3.2.

We divide the application design into two large-grain tasks to promote a new stakeholder: the context-awareness designer. The application designer produces the application model and classes. The context-awareness designer produces context-awareness models. Those models are built at design time in order to be manipulated at runtime and have to conform to a specific meta-model presented in Section 4.2. A context-awareness model of the illustrative scenario is given in the same section.

### 2.5. Overview of the meta-model views

We structure context-awareness data using three meta-models views, as shown in Figure 3. This enables us (i) to share context and collector models between several context-aware applications and (ii) to load several context and collector models coming from different sources.

The *context view* defines the observable and the interpreted observable concepts allowing them to be independent from applications. Thus, each observable model is then a catalog of pre-defined observables that context-awareness designers can use at any time. A context-awareness designer selects observables from one or several observable models which are relevant for an application and links the observables to entities. Those observables are designed to be reused by several context-aware applications.

The *collector view* defines the characteristics of each collector and the required information to utilize them in a context-aware application. Models of this view are designed to be shared by several applications. The dependence link with the context view is necessary because collectors are defined for observables.

The *context-awareness view* defines context-aware systems, it describes the entities to observe, the observables, the interpreted observables, the adaptation situations, and the different constrained contracts linked to each observable or adaptation situation. This view depends on elements described in the previous views.

Context-awareness modelling will be achieved through the definition of models (M1 level in MDE terminology) which conform to meta-models (M2 level). With these models, the

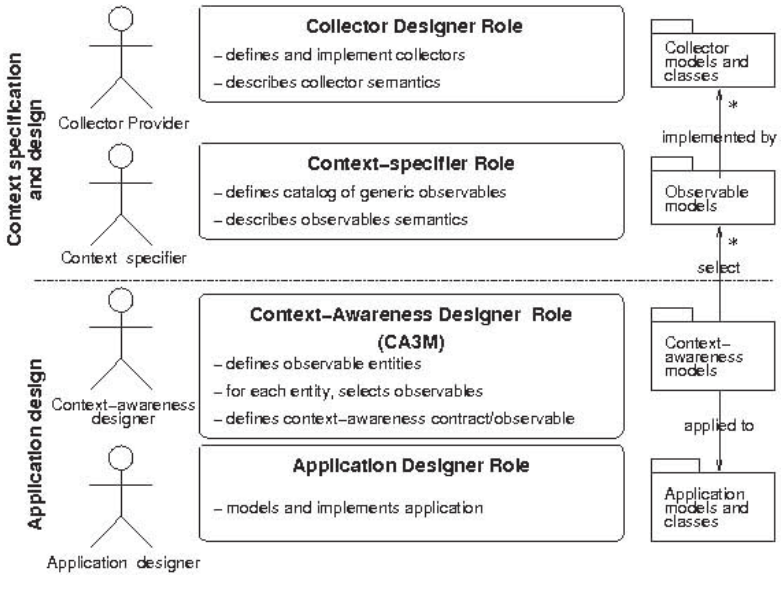


Figure 2. Separation of design tasks for producing context-aware applications

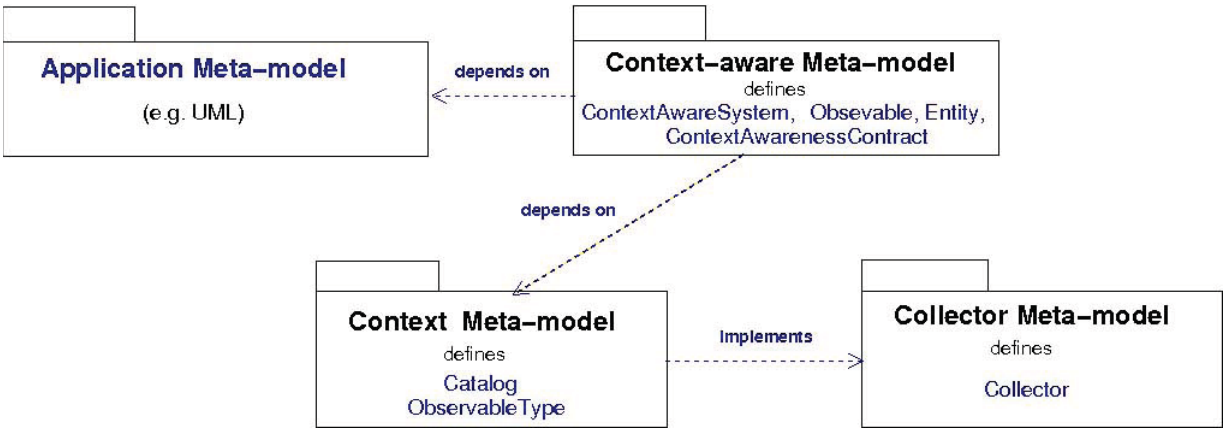


Figure 3. Relationship between meta-models

middleware can instantiate collectors to obtain data from the context environment (M0 -instance- level).

## 3. Context specification and design

We present, in this section, the context and collector meta-models, illustrated with the models for the observables extracted for the scenario. These observables will be used to construct the context-awareness model from the scenario that will be presented in Section 4.2. Context specifiers and designers define reusable observables and collectors. They define how high level observables (i.e., interpreted observables) may be computed from elementary observables.

### 3.1. Context specification

With the context meta-model (shown in Figure 4), context specifiers define catalogs of observables. Context models conform to this meta-model may be used by one or more context-aware systems and a context-aware system may use one or more of these models produced by several specifiers.

*ContextRoot* is the entry point of any context view model conform to this meta-model. ContextRoot is an aggregation of observable types.

In order to have a hierarchical view, we use the meta-class *Category* which allows us to better classify each observable type.

*ObservableType* attributes are: "name", "description", "immutable" (defines if the observable may change during runtime), "numerical", "observationJavaType" and "observationTypeName". At execution time, an observable type will result in the collection of one (for immutable) or several observations. "observationTypeName" and "observationJavaType" define the type of these observations.

*InterpretedObservableType* is an observable type which observations are obtained by applying a function that takes as entry parameters observations of a set of observables. The attribute "derivationExpression" expresses a derivation operation whereas the attribute "aggregation" is a boolean indicating if the result of this composite observation is an aggregation of elementary observations.

An *AdaptationSituationsType* is an interpreted observable type which represents changes in the state of the context information space. These significant changes require one or more system reactions called adaptations.

Figure 5 illustrates the context views related to the e-commerce application scenario. Some observable types (i.e. "BatteryLevel", "BatteryPlugged", "BatteryState", "TemperatureValue", "TemperatureState", "VideoPreference") are common to any

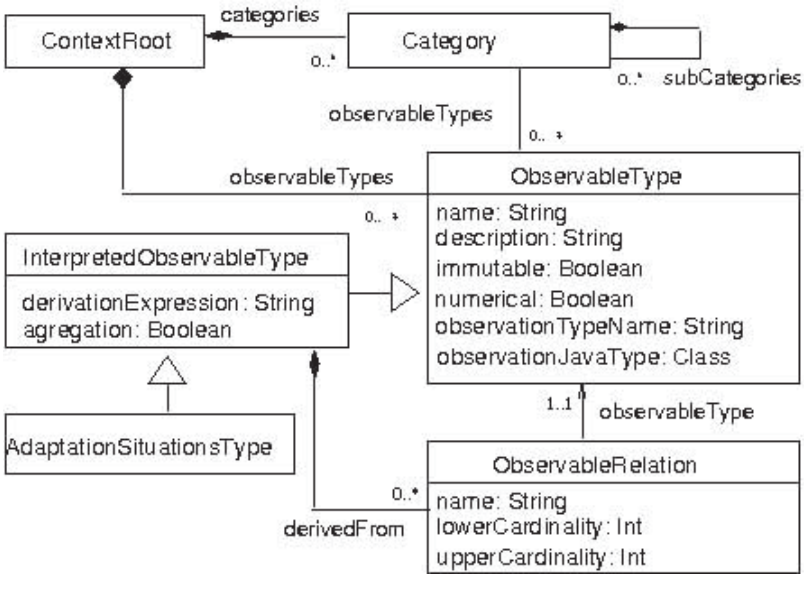


Figure 4. The context view meta-model

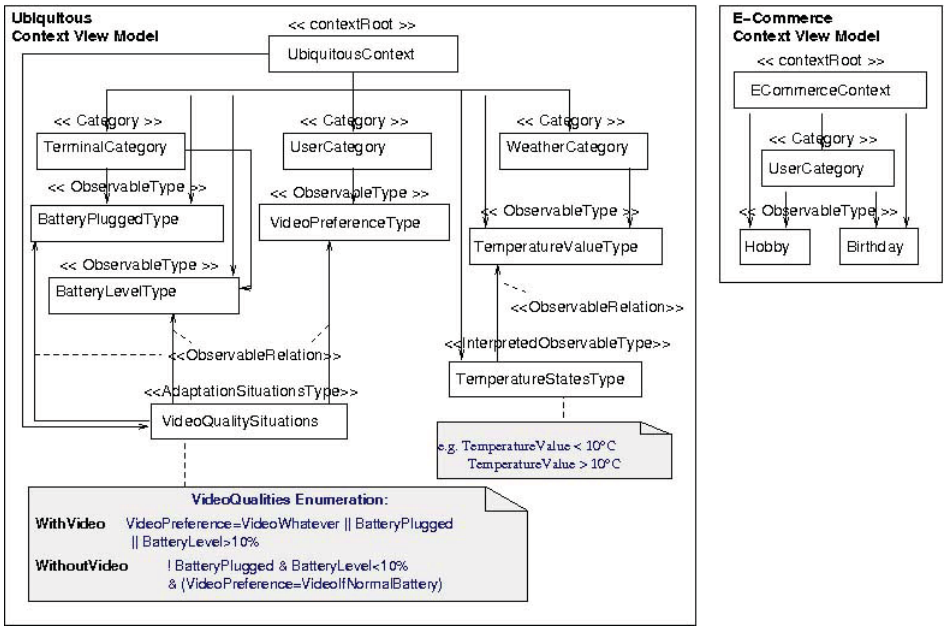


Figure 5. Context view models for the e-commerce scenario

ubiquitous application and thus are defined in a generic Ubiquitous context view model. We categorize hierarchically each observable type. For example, we classify "BatteryPlugged" and "BatteryLevel" in the same category *TerminalCategory*. For the observables which can not be found in general catalogs, new models may be defined for an application purpose. Thus, to illustrate our scenario, we add in Figure 5 the e-commerce Context View model. Here, we classify under the "UserCategory" the observables "Hobby" and "Birthday". Indeed, we need these two observables because the plug-in "Offers" uses Susan's hobbies and Betty's birthday and hobbies.

Some observable types may lead to adaptations, some of them are *AdaptationSituations*Type. "VideoQualitySituations" is an interpreted observable type derived from three source observable types: "BatteryPlugged", "BatteryLevel" and "VideoPreference". "VideoQualitySituations" defines two adaptation situations: "WithVideo" and "WithoutVideo". "WithVideo" state refers to the case where the user prefers to download videos whatever the battery state is (VideoPreference=Whatever _ BatteryPlugged _ BatteryLevel> 10%). "WithoutVideo" corresponds to the case where the user prefers to omit videos if the battery reaches its low level (VideoPreference=VideoIfNormalBattery & :BatteryPlugged & BatteryLevel< 10%). Many other observable types such as "ScreenSize" or "PicturePreference" exist in the model but are not represented here for clarity reasons. Computed together, these observable types lead to the adaptation situation "PicturePreference" with two possible values: "WithPicture" and "WithoutPicture".

### 3.2. Collector design

The collector design phase consists in providing concrete software for monitoring the environment. Software may be available for different context management frameworks such as (Baldauf et al., 2007; Coutaz et al., 2005; Dey, 2000; Paspallis et al., 2008; Conan et al., 2007). Information is necessary at the collector level in order that the context-aware system may (i) choose a collector among several collectors designed for a given observable type; (ii) produce the adhoc monitoring code to interface the context-aware system with a given software collector or alternatively use an adequate bridge for that collector;

We present the main classes of the collector meta-model in Figure 6 that we describe below. It defines meta-information necessary to use collectors.

*CollectorRoot* represents the entry point of any collector model conform to this meta-model. Collector is the main class of this meta-model, it defines meta-information necessary to use the collector. The signification of his main attributes are as follows. The *collectorFamily* attribute (e.g. COSMOS (Conan et al., 2007) collector) is necessary because each family may have its own rules to connect to collectors. There are two connection mode attributes: *notificationModeAvailable* and *observationModeAvailable*. If the notification mode is available, the collector is able to notify the context-aware system when there are significant modifications. With the observation mode, the context-aware system drives the observations. Both modes may be available for the same collector. An attribute (*unitOfMeasure*) defines the unit of measure of the collector (e.g. the number of minutes left or a percentage for a battery level observable type). Each collector is associated to one observable type. A collector may be attached to quality of context data (e.g. validity period, accuracy).

The *CollectorFactory* association is necessary for the system to concretely connect to a collector during the execution. A connection to the collector may be achieved through instantiation or dis- covery. In the instantiation mode, an instance of *instantiationArtifact* is created in the context-aware system. In the discovery mode, a connection to an existing collector is established.

## 4. Context-awareness system design

We believe that context-awareness is one preoccupation that should be taken into account during an application modelling process. We made the statement that usual modeling languages, such as UML, can not, as far as we know, express applications' context-awareness. Our goal is to overcome this limitation by offering designers the possibility to define application context-awareness models. In this section, we show how a context-awareness specific model is weaved with the application models and context models to configure application context-awareness.

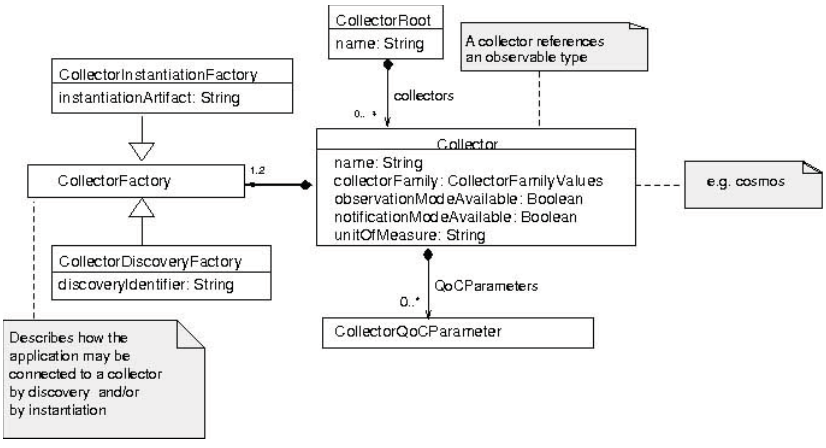


Figure 6. The collector meta-model

Firstly, we present in Section 4.1 an application meta-model and a simple example of the e-commerce application model. Secondly, in Section 4.2, we detail the context-awareness meta-model and the illustrative scenario context-awareness model.

### 4.1. An illustrative application meta-model and model

The context-awareness design process presented in this article is not targeted to a specific application modeling language. Several modeling languages such as a SCA modeling language (Service Component Architecture (Chappell, 2007)), UML (Object Managment Group, 2009) may be weaved with the context-awareness model. To illustrate a specific adaptation contract, we present in this section an example application meta-model. It is a simplified meta-model which may be used to define a UML component assembly.

An application model must be specified prior to the design of the application context-awareness model. The reason is, to define adaptations, the context-awareness designer needs to reference elements of the application model. Furthermore, adaptation contracts are targeted to a specific application meta-model. As adaptation contracts refer to application models, we foresee that it is necessary to define specialised adaptation contracts for each application meta-model. In Section 4.2, we illustrate an adaptation contract constructed for the simple meta-model presented in this section.

The example application meta-model is shown in Figure 7. The application model defined for the scenario and shown in Figure 8 is conform to this application meta-model.

We consider in this meta-model that an application is a set of components. Components may be connected together through their ports thanks to connectors. A component has a name and is of a given component type. A component type has provided and required interfaces. A component type may have several realisations (or implementations). For example, we consider to have several realisations, each one targeted for a given situation. We give in Figure 8 a part of the e-commerce application model conform to the above meta-model.

### 4.2. Context-Awareness meta-modelling and modelling

For the context-awareness preoccupation, we introduce a context-awareness meta-model to enable application designers to model their system context sensitivity. The designer is helped in his task with a specialized editor. In this sub-section, we firstly present the concepts included in this metamodel and shown in Figure 9. Then, we illustrate those concepts with the e-commerce scenario context-aware model depicted in Figure 10.

The context-awareness meta-model references both context view and collector view meta-classes. It may also reference application meta-classes.

*Context-Aware System* is the entry point of this meta-model. The left part of the meta-model defines the entities, the observables, the links between entities, the interpreted observables and the adaptation situations.

*Entity* represents a logical or physical element to be observed. For the scenario, we consider the user, the user's friend, the terminal and the cell's weather station entities. The *Entity* meta-class allows a context-aware system to differentiate several distributed observables. For example, we can obtain

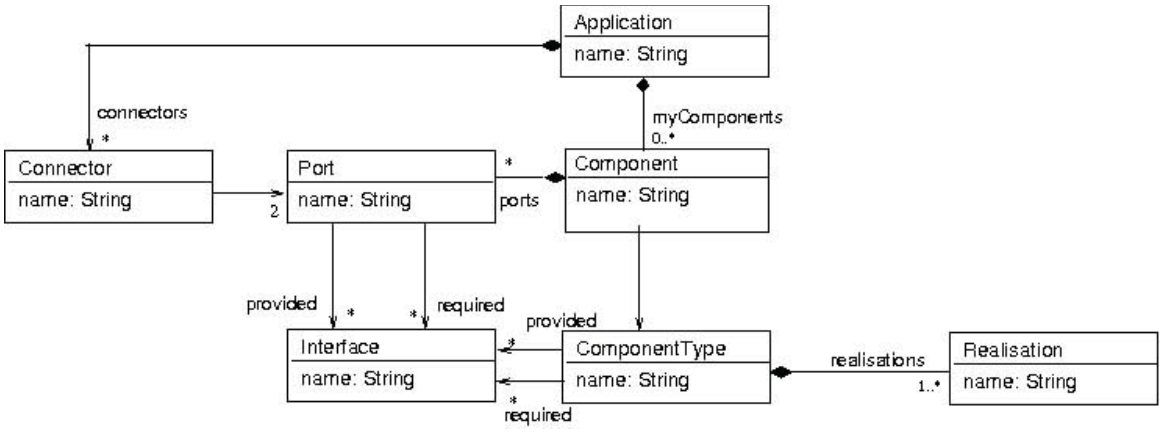


Figure 7. A simple application meta-model

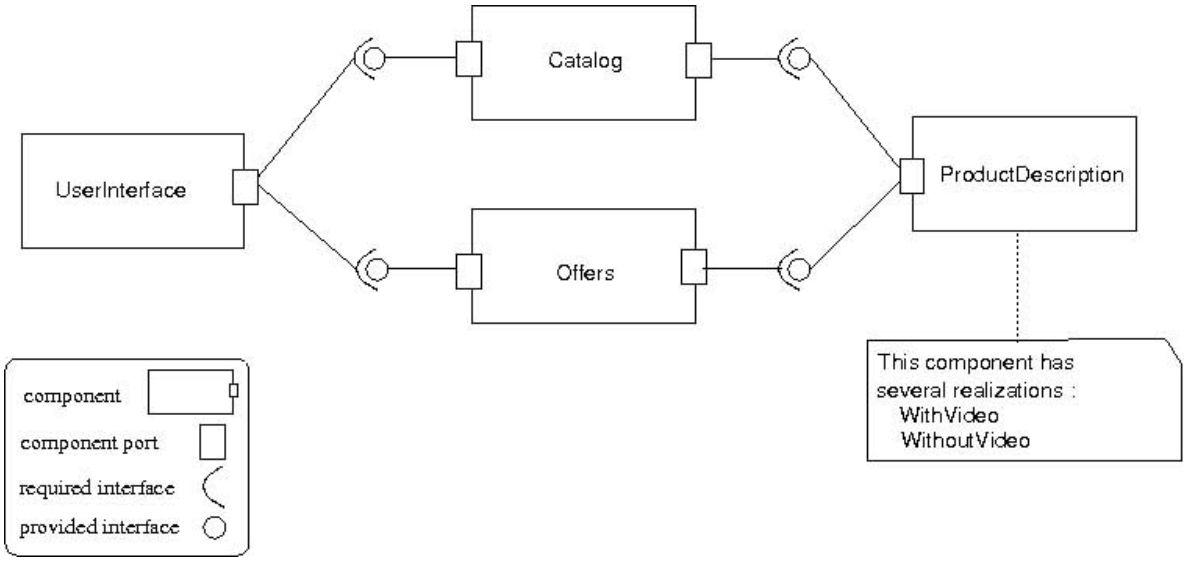


Figure 8. The application model

temperature observations from the cell "WeatherStation" entity, or from the server "WeatherStation" entity. As these two entities may be distant from thousand of kilometers, the observations may be quite different. The entity concept allows the designer to express both of them. An entity may be linked to another entity through the *EntityRelation* meta-class. For example, we link the entity "User" to the entity "Terminal". These associations may be meaningful to identify observables in the context-aware system.

An *Observable* meta-class references the *ObservableType* meta-class of the context meta-model. For each entity, the designer chooses the type of their observables in the catalog of observable types defined in pre-loaded context models.

When the designer defines an *InterpretedObservable*, after choosing its type in the observable type catalogs, the designer is asked to choose the source observables (defined by the *derivedFrom* association of Figure 4). With the context model, the context-awareness editor may verify the type of the source observables. The context-awareness designer is asked for the entities to which are linked the source observables. For example, when defining the “VideoQualities” observable of Figure 10, the designer is asked to to choose a “VideoPreferenceType” observable as well as a “BatteryLevelType” and a “BatteryPlugged” observables.

The right part of the meta-model defines three kinds of contracts: *Observation Contract*, *Notification-Contract* and *Adaptation Contract*. The filled pattern part of the meta-model shows the meta-classes which are dependant of the application meta-model. A *context-awareness contract* defines a contract between an observable and an application. In an *Observation Contract* is expressed the quality of context required by the application (*e.g.,* accuracy, unit of measure). With an *Observation Contract*, the application designer defines that the application needs these observables and that the application will make active requests to the collector to obtain observations.

With a *Notification Contract*, the application designer defines in addition that the application subscribes to events. These events happen for example when a numerical observable value

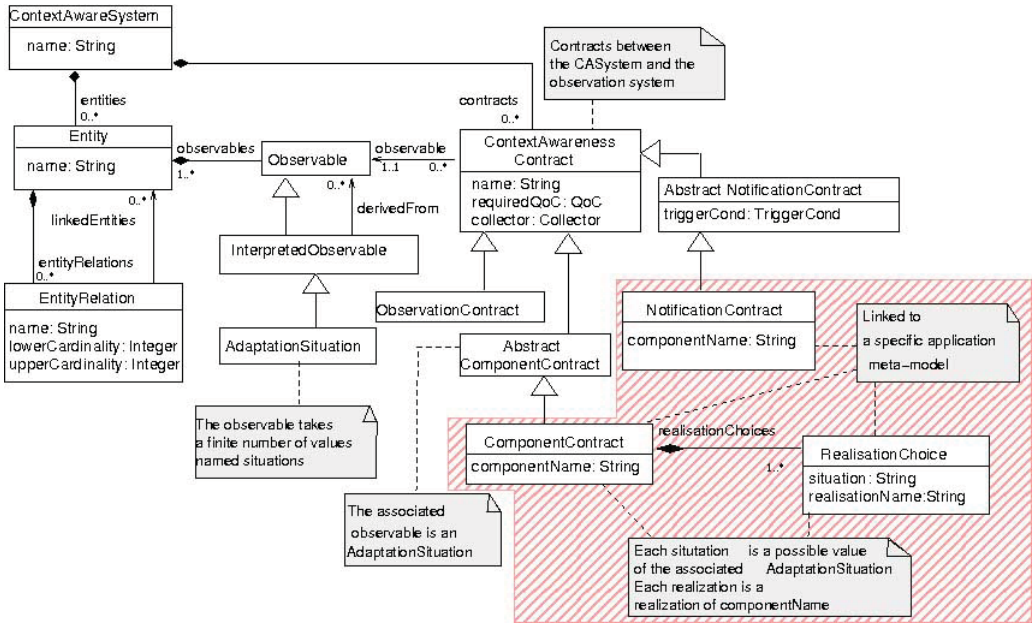


Figure 9. The context-awareness meta-model

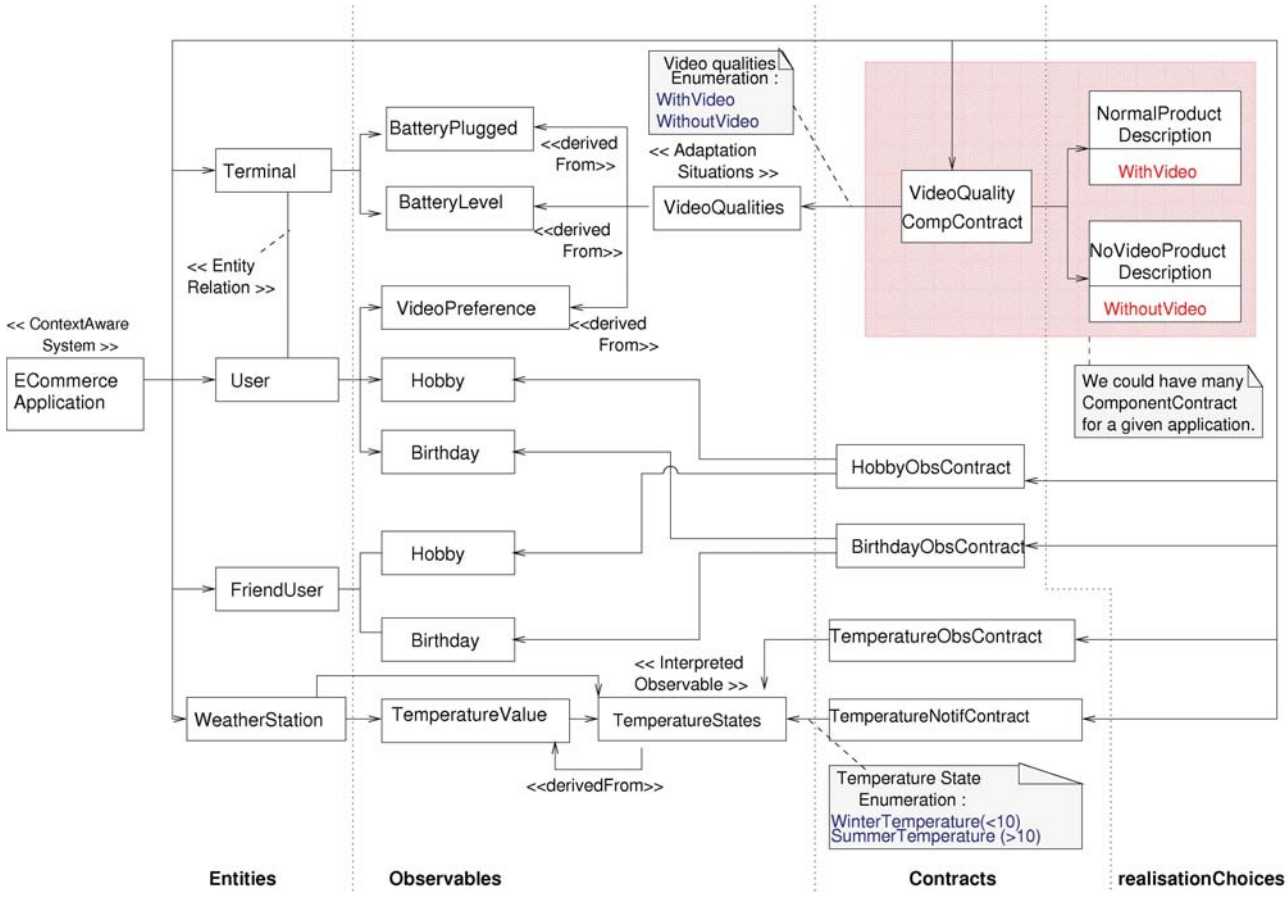


Figure 10. The application context-awareness model

reaches a fixed threshold or when an enumerated observable value changes from one enumeration to an other, the condition is defined with the "triggerCondition" attribute. The designer is then asked to choose

the application element, such as the concerned application component name. However with notification contracts, the adaptation decisions are left to the application part.

Some adaptation decisions (*i.e.,* the choice of an application variant) can be defined within a context-awareness model. This is possible thanks to the *Adaptation Contract*, in particular the *Component-Contract*. With this kind of contract, the context-awareness designer defines for a given component a set of correspondences between an adaptation situation and the component realisation name.

For our scenario, we define two observation contracts and a notification contract. Indeed, the "Offers" application component needs at most three kind of observations inputs to be context-aware: Suzanne's "hobby", Betty's "birthday" and the outside "temperature". Note that these observables are collected from different entities which may also be distributed. The "Offers" application component connects to the context-awareness via the *Observation Contract* as shown in Figure 10. We choose to link the "temperature" interpreted observable with another contract (*Notification Contract* ) in order for the application to be notified if the temperature state enumeration changes.

In the scenario, the adaptation concerns the choice between two realisations of the "Product Description" component: "Video Product Description" and "WithoutVideo Product Description". The possible values of the adaptation situation "VideoQualities" (computed from its source values) can be (VideoPreference=Whatever $\vee$ BatteryPlugged $\vee$ BatteryLevel> 10% ) and `WithoutVideo` ( VideoPreference=VideoIfNormalBattery & ¬BatteryPlugged & BatteryLevel< 10% ). According to the evaluation of these adaptation situations, the Adaptation Contract leads to one of these two component realisations ("WithVideo Product Description" realisation is the variant chosen in case of `WithVideo` "VideoQualities" adaptation situation and "Without Product Description" is the variant chosen in case of `WithoutVideo` "VideoQualities" adaptation situation). If no state corresponds to the current situation, a default system model variant may be instantiated through the *Adaptation Contract*.

## 5. Implementation choices

With respect to MDE approach, MOF (*Meta-Object-Facility*) (Object Management Group, 2006), ECORE from EMF (*Eclipse Modeling Framework*) (Budinsky *et al.*, 2008) and UML Profile (Object Managment Group, 2009) are the most popular meta-modeling languages. For our implementation, we made the choice of EMF technology for the two reasons. The first one is that we have eliminated UML profile because it does not allow designers to define associations between profile meta-classes. The second reason is that we choose ECORE instead of MOF for the availability of the EMF tools. Compared to MOF, ECORE lacks the possibility to define meta-associations. To overcome this limitation, we define special meta-classes (such as *EntityRelation* in the context-awareness view) to be used in place of meta-associations when necessary.

With EMF, we have specialized model editors. One editor for context view model, one editor for collector view model, one editor for application view models and one editor for context-awareness view models. Editors weave links between the different models and make verifications (*e.g.,* verification of source observable types when defining an interpreted observable when editing the context-awareness model).

We can use ECORE models for transformation purpose to generate application context-awareness code. We can also load the models at runtime. The EMF generated APIs for each meta-model is used for navigation between the model elements. With the model loaded at runtime, new collectors and context-awareness mechanisms may be added during runtime. Thanks to EMF adaptors positioned in the model, reactions to modifications in the model may be triggered. This part of the work is not presented in this article.

With specialized editors, context-awareness meta-model helps designers to define the application context-awareness preoccupation. The context-awareness may be reconfigured by updating the model. A complete chain from context-awareness model to application runtime context-awareness may be designed with MDE technologies.

## 6. Related work

Due to the variety of context to be collected and analyzed, context management needs the support of abstract context modelling. Main families of context modelling are profiling (e.g. CC/PP (Klyne *et al.*, 2007)), data-bases (*e.g.,* CML (Henricksen *et al.*, 2006)), ontologies (*e.g.,* CONON (Wang *et al.*, 2004)) and MDE. Our work aims at using MDE for defining links between context modelling to express complex context situations, and context-awareness modeling to link context situations to application variations.

In this section, we present some projects which deal with meta-modelling of context-awareness using the MDE approach. The projects we study here are ContextUML (Sheng *et al.*, 2005), CAPPUCINE (Parra *et al.*, 2009), Scatter (White *et al.*, 2008), and Ayed (Ayed *et al.*, 2007).

*ContextUML* (Sheng *et al.*, 2005) was one of the first domain specific model for context-awareness. It defines a meta-model for modelmodellingling context-awareness of web services. Consequently, web services elements such as *Service*, *Operation* and *Message* are represented in the model as well as related adaptation mechanisms of type *Binding* or *Triggering*. A binding mechanism defines a relationship between an observable and an element of the application model. Operations parameters may receive context observations using this binding. The triggering mechanism associates an action to an adaptation situation. These actions are filtering / processing operations applied to input or output messages of a web service.

The approach we follow in our solution is similar to Context UML. However, we differ in the following points. First, we plan several model views. Secondly, we introduce the concept of entities. Finally, Context UML meta-model is supposed to be used with a web service meta-model. Our context-aware meta-model may be used with any ECORE application model. It may be a UML model available in ECORE as well as a web service model with an available ECORE model. Each

kind of model diagram may be extended with context-awareness. For this purpose, middleware or transformation processes to handle these descriptions have to be defined.

CAPPUCINE (Parra *et al.*, 2009) describes an MDE approach for dynamically producing product lines according to context information. CAPPUCINE and ContextUML put the stress on adaptation mechanisms rather than context modelling. Our work, enable application designers to express complex situations computed from distributed context observations. Furthermore, it adds a collector preoccupation which allows to link several context manager frameworks to the application.

*Scatter* (White *et al.*, 2008) provides the means to define the variations of a given application according to the run-time environment. These variations are defined using FODA (*Feature-Oriented Domain Analysis* (Cohen *et al.*, 1998)) diagrams to define application and platform features. Application features define their requirements on platform features. A constraint solver computes at application deployment time a product line derivation. With Scatter, the only observable entity is the terminal. The application life-cycle concerned by context -awareness is the deployment. The computed product line is un-determinist. The advantage is that the resolution is flexible; the disadvantage is that Scatter can not ensure that a solution will be computed and which one.

Ayed (Ayed *et al.*, 2007) defines a process to integrate context-awareness in UML application modelling. The process is defined in six steps which: (*i*) define observables, (*ii*) define application context-awareness, (*iii*) define collectors, (*iv*) define abstract platform model, (*v*) define model transformation for concrete platform, (*vi*) produce code generation. Steps (*i*), (*ii*), (*iii*) are defined with UML profiles. The advantage of defining context-awareness modelling through UML profile is to ease the integration of context-awareness definitions in standard UML tools. With our solution, a specific tool which integrates context-awareness with context modelling has to be provided. One disadvantage of their solution is due to UML profile limitation which does not allow profile designers to define associations between profile meta-classes. This limitation, for example, does not allow the profile designer to link observables to entities.

## 7. Conclusion and future work

In this article, we have presented meta-models for defining context-aware application models. We have followed the MDE approach. MDE approach provides a high level of abstraction. The advantage is that models may be applied for different platforms and technologies especially different context management technologies. We argue that code which manages context-awareness may be automatically produced from context-aware models. This relieves developers from context-awareness implementations and also allows designers to easily modify context-awareness configuration during the whole application life-cycle.

We propose three steps for obtaining context-awareness configurations. First of all, the context view step allows context-awareness designers to benefit from the definition of general purpose observables which may be collected through several collectors. The collector step has to be defined by any context manager provider which offers a sensor or a context computing unit. The context and collector abstractions allow applications to be connected to various kinds of collectors. The link between an observable and a concrete collector may be decided at runtime. The collector may then be discovered or instantiated depending on the available collectors. Finally, applications may define their own context-awareness. We provide several context-awareness contracts kinds: observation, notification and adaptation contracts. We have shown how a context-awareness contract may be linked to an application model, several notification and adaptation contracts may be added to define new kind of adaptations in the application. Meta-models allow to specify the verifications which have to be done at different steps and especially when defining a new contract in a model.

We evaluate the meta-models we have presented here on several scenarios and applications. We are currently developing a context-aware middleware. This middleware loads at runtime a

context-awareness model and instantiates the necessary collectors for this model. It also takes in charge notification tasks that have been defined in the model.

For each kind of application model (e.g. UML class diagram, UML sequence diagram, SCA components) new context-awareness contracts have to be defined, modelling editors have to be specialised for them and new context-awareness middleware tools have to be developed to automate context-awareness management.

**References**


[1] Ayed, D., Delanote, D., Berbers, Y. (2007). *Computer Science*, V. 4635/2007 of *Lecture Notes in Computer Science*, Springer, Berlin / Heidelberg, chapter MDD Approach for the Development of Context-Aware Applications, p. 15-28.

[2] Baldauf, M., Dustdar, S., Rosenberg, F. (2007). A Survey on Context-Aware Systems, *International Journal of Ad Hoc and Ubiquitous Computing*, V. 2 (4) p. 263-277.

[3] Budinsky, F., Merks, E., Steinberg, D. (2008). *Eclipse Modeling Framework 2.0*, Eclipse, Addison Wesley Professional, March.

[4] Chappell, D. (2007). Introducing SCA, white paper, Chappell and Associates, July.

[5] Cohen, S., Northrop, L. (1998) Object-Oriented Technology and Domain Analysis, *In:* Fifth International Conference on Software Reuse, Los Alamitos, CA, p. 86-93, June.

[6] Conan, D., Rouvoy, R., Seinturier, L. (2007). Scalable Processing of Context Information with COSMOS, *In:* Springer-Verlag (ed.), 7th IFIP International Conference on Distributed Applications and Interoperable Systems, V. 4531 of *Lecture Notes in Computer Science*, Paphos, Cyprus, p. 210-224, June.

[7] Coutaz, J., Crowley, J., Dobson, S., Garlan, D. (2005). Context is Key, *CACM*, V. 48 (3) 49-53, March.

[8] Dey, A. (2000). Providing Architectural Support for Building Context-Aware Applications, PhD thesis, College of Computing, Georgia Institute of Technology, December.

[9] Henricksen, K., Indulska, J. (2006). Developing context-aware pervasive computing applications: Models and approach, *Pervasive and Mobile Computing*, V. 2 (1) p. 37-64, February.

[10] Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M. H., Tran, L. (2007). Composite Capability/Preference Profile (CC/PP): Structure and vocabularies 2.0, Technical report, W3C recommandation, April.

[11] Object Management Group. (2006). Meta Object Facility (MOF) Core Specification Version 2.0. , OMG document formal/06-01-01, January.

[12] Object Managment Group. (2009). UML 2.2 Superstructure Specification, OMG documents formal/2009-02-02, February.

[13] Parra, C., Blanc, X., Duchien, L. (2009). Context -Awareness for Dynamic Service-Oriented Product Lines, 13th International Software Product Line Conference (SPLC), San Francisco, CA, USA, August.

[14] Paspallis, N., Rouvoy, R., Barone, P., Papadopoulos, G., Eliassen, F., Mamelli, A. (2008). A Pluggable and Reconfigurable Architecture for a Context-aware Enabling Middleware System, *Proc. 10th*, V. 5331, Monterrey, Mexico, p. 553-570, November.

[15] Schilit, B., Theimer, M. (1994). Disseminating Active Map Information to Mobile Hosts, *IEEE Network*, V. 8 (5) p. 22-32.

[16] Sheng, Q., Benatallah, B. (2005). ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services, *In:* The 4th International Conference on Mobile Business (ICMB'05), IEEE Computer Society. Sydney, Australia., p. 206-212, July 11-13.

[17] Wang, X. H., Zhang, D. Q., Gu, T., Pung, H. K. (2004). Ontology Based Context Modeling and Reasoning using OWL, *In:* 2nd IEEE Conference on *Pervasive Computing and Communications* (PerCom2004), Orlando, FL, USA, p. 18-22, March.

[18] White, J., Schmidt, D., Wuchner, E., Nechypurenko, A. (2008). Automatically Composing Reusable Software Components for Mobile Devices, *Journal of the Brazilian Computer Society Special Issue on Software Reuse SciELO Brasil*, V. 14 (1) p. 25-44, March.