Context-awareness and Model Driven Engineering: Illustration by an E-commerce application scenario

Chantal Taconet and Zakia Kazi-Aoul
Institut TELECOM; Telecom et Management SudParis; CNRS SAMOVAR
9 Rue Charles Fourier, 91011, Évry Cedex, FRANCE
Chantal.taconet@it-sudparis.eu, Zakia.kazi-aoul@it-sudparis.eu

Abstract

With the popularity of ubiquitous computing, context-aware applications become clearly necessary. This kind of applications allows mobile users to universally access services in respect to any user context including his computing environment. Challenges for these applications are to easily manipulate both context collection and analysis and to react dynamically to every relevant evolution in the execution context state. To face these issues, we propose in this article, a generic and extensible way to model context-awareness of any application using model-driven engineering (MDE) approach. We illustrate our solution by modeling a context-aware E-commerce application.

1. Introduction

One consequence of ubiquitous computing is that applications may be used in a wide variety of environments. For this purpose, context-aware applications emerge. One of the first use of the term context-aware applications appeared in 1994 [11]. According to Dey’s definition [5], 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 terminal battery level and connectivity.

In the past years, many services have been designed to manage the context among them ContextToolkit [5], CONON [13], COSMOS [4]. 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 context modifications relevant for context-aware applications are detected). One issue for context-aware applications is to easily make use of these various services both at a low development cost and with easy reconfigurations enablers.

For context-aware application designers, the main difficulties come from the multiple kind 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 the portability: the context awareness implementation may easily makes use of new context management services.

We propose in this article a generic and extensible way to model context-awareness in an application using meta-models. The next issue would be to integrate context-awareness design in the application modeling process.

The next section introduces an E-commerce illustrating scenario. Section 3 presents and describes the meta-models illustrated by the E-commerce application scenario. In Section 4, we present some related work, and finally we conclude in Section 5.
2. E-commerce illustrating scenario

We present in this section a scenario that concerns an ubiquitous application.

Suzanne is a client of a famous E-commerce merchant where she often makes purchases of all kinds. Suzanne’s client profile may be 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 site. When connected, Suzanne receives "offers" on (i) hiking shoes because Suzanne’s hobby is hiking and (ii) pullovers because the "temperature" measured by the weather station near the current mobile cell is 5 degrees Celsius. Just when Suzanne decided to look at the products in detail using the application component product description, the battery level reaches its low level. So the application switches to a poor mode omitting images and animations. 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 face changes in Suzanne’s context.

3. Meta-models

We believe that context-awareness must be defined during an application modeling process. Starting from the observation that usual modeling languages, such as UML, can not, as far as we know, express the applications context-awareness, our goal is to propose a context-awareness language which would be mixed with the application model.

The context-awareness modeling will be achieved through the definition of models (M1 level in MDE terminology) which conform to meta-models (M2 level). With these models, the middleware can thus instantiate collectors to obtain data from the context environment (M0 - instance-level).

3.1. Terminology

In order to better understand the concepts shown in Figure 1, we first need to define some elementary concepts.

**Entity** An 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 has one or more collectors.

**Adaptation situation** An adaptation situation represents changes of state of an observable value which may have for result a modification (adaptation) in the context aware system. An adaptation situation is, for example, Suzanne’s terminal battery state which value could be "LowBattery" or "NormalBattery".

3.2. Meta-model views

We structure context-awareness data using three meta-models views, as shown in Figure 1. 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 contains observables. It may be defined independently from the application and collector models. Observables are dedicated to be reused by several 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.

3.3. Implementation choices

With respect to MDE approach, MOF (Meta-Object-Facility) [8], ECORE from (Eclipse Modeling Framework) [2] and UML Profile [9] are the most popular meta-modeling languages. For our implementation, we made the choice of EMF technology for the following reasons.
First of all, we have eliminated UML profile because it does not allow designers to define associations between profile meta-classes.

Finally between MOF and ECORE we have chosen ECORE for the availability of the EMF tools. Compared to MOF, ECORE lacks the possibility to define meta-associations. As a consequence, we define special meta-classes to be used in place of meta-associations when necessary.

We edit meta-models with the EMF ECORE editor. We edit models conform to these meta-models with the generated EMF ECORE editors.

We could use ECORE models for transformation purpose to generate application context-awareness code. We prefer instead to write middleware utilized at execution time. At execution time, context-awareness models are loaded and computed through EMF generated APIs. We can then discover or instantiate collectors. The application may ask for observations. The middleware can react to context notifications. New collectors and context-awareness mechanisms may be added during the application execution thanks to EMF adaptors positioned in the model. The middleware is not presented in this article, we present rather the meta-model in the following parts of this section.

3.4. Context meta-model

This meta-model is an abstraction 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.

![Figure 2. The context view meta-model](image)

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" (a boolean which determines if this element is immutable or not), "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. These reactions are modeled in the context-awareness model (c.f. Section 3.6).

![Figure 3. Context view models for the E-commerce scenario](image)

Figure 3 illustrates the context view related to the E-commerce application scenario. Some observable types (i.e. BatteryLevel, BatteryPlugged, BatteryState, TemperatureValue, TemperatureState) are common to any ubiquitous application and thus are defined in an Ubiquitous Context View model.

We categorize hierarchically each observable type. For example, BatteryPlugged and BatteryLevel are classified in the same category TerminalCategory.

The observable type Hobby is significant for E-commerce applications only and is defined in the E-Commerce Context View model. Many other observable types such as Memory, PreferredProduct or Humidity exist in the model but are not represented here for clarity reasons. Some observable types may lead to adaptation situations. BatteryState defines two adaptation situations (i.e.
NormalBattery and LowBattery). BatteryState is an interpreted observable type derived from BatteryPlugged and BatteryLevel observable types.

### 3.5. Collector meta-model

Models conform to this meta-model may be used by one or more context-aware system models. Data stored in these models may be used during the execution in order to collect observations or to be notified of observation significant changes. We show the main classes of the collector meta-model in Figure 4 that we describe below.

**CollectorRoot** represents the entry point of any collector model conform to this meta-model.

**Collector** is the main class of this meta-model. The significance of his main attributes are as follows. The **collectorFamily** attribute (e.g. COSMOS [4] 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 notifies the context-aware system when there are significant modifications. If the observation mode is available, 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. observable type).

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.

### 3.6. ContextAwareness meta-model

The goal of the context awareness meta-model is to enable application designers to model their system context sensitivity. The concepts included in this meta-model are illustrated both by the context-aware system meta-model presented in Figure 5 and the context-aware system model defined for the E-commerce scenario presented in Figure 6.

The meta-model references both context view and collector view meta-classes. As we will see later, it may also reference external application model elements defined in a standard modeling language such as UML or SCA assembly language [10]. The SCA assembly language defines an assembly of components. Each component describes its required and provided services. Each service is defined by its operations.

**ContextAwareSystem** 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 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 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, the entity “User” could be linked to the entity “Terminal”. These associations may be meaningful to identify observables in the context-aware system.

An **Observable** meta-class represents an observable connected to a concrete collector which collects and/or notifies observations.
An InterpretedObservable and an AdaptationSituations are respectively linked to an InterpretedObservableType and an AdaptationSituationsType (meta-classes presented in the context view meta-model). This is possible because the context-awareness view meta-model references the context view meta-model. For example, the interpreted observable "TemperatureStates" (which is obtained from the observable "TemperatureValue") is of type "TemperatureStatesType".

The right part of the meta-model defines two kinds of contracts. The goal of these contracts is to ease the adaptation or the selection of external application model elements. This application may be for example a SCA hierarchical composite which includes several components.

The context-aware system could be a line products application. The line product chosen will be conditioned by the context as we will see in the definition of the AdaptationContract.

We define two kinds of impact of the context to the application behavior: updates and variations. Updates are triggered through ObservationalContract whereas variations are selected thanks to AdaptationContract.

The first impact is to update, at the instance level, an application element or property. For our scenario, two updates are defined. The first update concerns the hobby attribute of the Offers application component. This update happens when the hobby collected from a general profile file changes. The ObservationalContract could be: when there is a hobby modification notification then trigger the operation Offers.setHobbyAttribute(CurrentValue(hobby)). The second update concerns the temperature state attribute of an application SCA component when the temperature collected from the cell’s weather station changes of state (< 10 ° or > 10 °).

The second impact is to define variable parts in the application model (i.e. a point in the model to which may be plugged different subsets of the models -i.e. product lines- according to the context state).

In the scenario, the "Product Description" component may have two sub-components "Normal Product Description" and "Poor Product Description". According to the evaluation of the adaptation situations, the AdaptationContract leads to one of these two sub-components ("Normal Product Description" sub component is the variant chosen in case of LowBattery temperature state and "Poor Product Description" is the variant chosen in case of NormalBattery temperature state). If no state corresponds to the current situation, a default system model variant may be instantiated through the AdaptationContract.

4. Related work

Due to the variety of context to be collected and analyzed, context management needs the support of abstract context modeling. Main families of context modeling are profiling (e.g. CC/PP [7]), data-bases (e.g. CML [6]), ontologies (e.g. CONON [13]) and MDE.

In this section, we present some projects which deal with meta-modeling of context-awareness using the MDE approach. The projects we study here are ContextUML [12], Scatter [14] and Ayed [1].

ContextUML [12] defines a meta-model for modeling 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 ContextUML. However, we differ in the following points. First, we plan several model views. Secondly, we introduce the concept of entities. Finally, ContextUML 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.

Scatter [14] provides the means to define the variations of a given application according to the runtime environment. These variations are defined using FODA (Feature-Oriented Domain Analysis [3]) diagrams to define appli-
cation 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 [1] defines a process to integrate context awareness in UML application modeling. 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 modeling 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 modeling has to be developed for each modeling diagram. One disadvantage 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.

5. 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 to model context-awareness in different steps. First of all, the context view level allows context-awareness developers to benefit from the definition of general observables which may be defined by context sensor providers and completed if necessary by application developers. The collector view has to be defined by any context manager provider which offers a sensor or a context computing unit. This view allows applications to be connected to any kind of collectors. Finally, applications may define their own context awareness. We provide two context aware mechanisms: updating and variations. Context awareness may be linked to any application model at the condition that context awareness semantics has been defined.

The meta-models we have presented here have been evaluated on several scenarios and applications. We are currently developing a middleware for connecting applications to COSMOS collectors and to implement context-awareness reactions for SCA applications. For each kind of application model (e.g. UML class diagram, UML sequence diagram, SCA components) the context awareness semantics has to be defined and modeling tools have to be provided.

References


