UNIVERSITÀ DEGLI STUDI DI ROMA "TOR VERGATA"

FACOLTA' DI INGEGNERIA INFORMATICA

DOTTORATO DI RICERCA IN
INFORMATICA ED INGEGNERIA DELL'AUTOMAZIONE

CICLO DEL CORSO DI DOTTORATO
XXI

Titolo della tesi
Model Driven Development of Context Aware Software Systems

Dottorando: Andrea Sindico

A.A. 2008/2009

Docente Guida/Tutor: Prof. Vincenzo Grassi

Coordinatore: Prof. Daniel P. Bovet

Abstract

Since the first computing machine up today, computers have passed from huge rooms of few military and government laboratories to the desks and pockets of common people. In this ever growing process many things have changed and are still changing the way we experience computers in our day by day life. New shapes have made them more comfortable and portable while new technologis have strongly enhanced the way man and machines interacte. Meanwhile, while computers spread out all over the world, new wired and wireless transmission technologies continuously connect them in a world wide network ever surrounding all of us. We are thus quickly moving toward the realization of the Mark Weiser's vision of ubiquitous or pervasive computing, described for the first time in 1991 [1], whose essence is the creation of an environment saturated with computing and communication capabilities, yet gracefully integrated with human users [2].

If in 1991 pervasive computing was only a vision too far ahead of its realization, in 2001 M. Satyanarayanan [2] claimed that because of the incredible hardware progress of the last decades, many critical elements of pervasive computing have become available commercial products, such as: handled and wearable computers; wireless LANs; devices to sense and control appliances. We are now close to the beginning of the second decade of the XXI century thus what are the missing capabilities we need to finally achieve this goal? For Satynarayanan the real research on pervasive computing is now related to find the right way to seamless integrate component technologies into a complex, but transparent to the user, distributed and highly dynamic system.

From a different point of view this is like to say that hardware technologies have made their part by providing affordable components which may support pervasive computing. It is now up to software engineers to introduce standard and structured solutions aimed at filling the gap with this objective. In fact such a new paradigm introduces new problems and challenges never faced before by software technologies, i.e.: resources management in heterogeneous and highly dynamic environments; continuous and dynamic system reconfiguration depending on the kind of the currently available resources; security issues related to the reliability of the available resources; etc..

All these concerns can be summarized in a single capability which ubiquitous pervasive systems should provide: context awareness. Context awareness can be unformally defined as the capability of a system to be cognizant of its users' state and surroundings, and able to modify its behaviors based on these information. Context

awareness is critical for mobile and ubiquitous computing, where devices must adapt their behavior to the current environment. However, despite the fact context is clearly a central notion to an emerging class of applications, there is yet poor explicit support for context awareness in mainstream programming languages and runtime environments; thing that makes the development of these applications more complex than necessary. The lack of accepted standards in context awareness modeling and development is due to the number of issues that still have to be addressed in this new research area. For example: how is context represented internally? How is this information combined with system and application state? Where is context stored? What are the relevant data structures and algorithms? How frequently does context information have to be consulted? what is the overhead of taking context into account? and What techniques can be used to keep this overhead low? Is historical context useful?

With the aim to provide a contribute toward a major comprehension of the context awareness issues in software engineering, we present a modeling framework that can be used by engineers to model systems' context aware characteristics independently of the possible implementation they could have. Our framework allows the designer to be focused on the information related to the entities involved into the realization of a context aware behaviour making easy to share his/her understanding of such concern with other designers or developers. Moreover our modeling approach is based on aspect oriented modeling techniques so that it is possible to model context aware behaviours for already existing systems without having to modify their original models but only referring to their elements.

Models based on this framework can be therefore object of transformation processes aimed at producing usefull artifacts such as: metrics or other measurments giving the designer feedbacks about his/her desigin choices; documentation which can be shared with the system's stakeholders; code which actually implements them.

Table of Contents

1 Introduction

The continuous process of hardware miniaturization together with the technological improvements in the field of energy management is leading to the born of mobile ubiquitous computing scenarios in which the computation is pervasive and possibly hidden to the user.

As a consequence of this trend, software engineers have to face new challenging problems in the design of systems that must adapt their behavior dependently of the continuous environmental changes and the specific user's characteristics. As an example consider the scenario described in [3] where a traveler service application aids the travelers to easily achieve basic services such as ticket purchase; automatic check-in or finding a restaurant in an airport. These services may be improoved by the introduction of some context dependent enhancement. For example, while looking for a restaurant the application could take into account the *cuisine* preferences of the user which can be themselves provided by existing user profiling services. Moreover the returned list of candidate restaurants could be arranged according to how much time the traveler may presumably spend at the restaurant, which could itself depend on whether s/he has successfully checked-in, and how much time remains for the scheduled flight departure. If the traveler has few time to spend, fast food restaurant could be displayed first, or even a warning could be given if the boarding time is approaching.

The ability of a system to change its behaviors depending on the characteristics of the environment is called *Context Awareness* (CA). The design and implementation of such a characteristic can be very tricky because of its tendency to crosscut the other concerns of the system in which it has to be introduced. Its crosscutting nature makes hard to encapsulate context aware behaviours into well separated components and thus to reuse them into different systems [3,4].

Several approaches have been proposed to address these issues during the design phase of a system, with context oriented modeling [5,6,7,8,9,10], or at the coding phase, with context oriented programming [11,12,13,14,15]. However these emerging technologies still lack a common agreement on what kind of information should make a context and how it can be modeled implemented and then exploited in order to change the system behaviors. As a consequence modeling and coding standards for context awareness are far to be defined thus slowing down the technological improvement that is actually needed for the realization of the ubiquitous computing vision.

Aimed at providing a contribute toward a major comprehension of the context awareness issues in software engineering, in this thesis we present a modeling framework that can be used by designers to model context aware characteristics independently of the possibly infinte ways they can be implemented but focusing on the information related to the involved entities. One of the main feature such a framework would provide is the possibility for the designer to quickly develop models capturing his/her understanding of the context awareness concerns and eventually making easier to share them with the other system's stakeholders.

For the realization of such environment we started from a comparison of the already existing context awareness modeling and development approaches in order to identify the common characteristics and most expressive solutions which have already been proposed in literature. In the 2nd chapter of this thesis we briefly summarized this part of the work also providing definitions for the "*context*" and "*context aware*" concepts.

Trying to take the best from the already proposed approaches, we have defined a *conceptual domain model* (CDM) [16] consisting of the conceptual constructs and relationships we think are needed to cover context awareness concerns. In the 3rd chapter we describe the designed CDM also providing a brief comparison between it and the existing modeling approaches described in chapter 2. This comparison is aimed at providing an informal demonstration of how our CDM can encompass the existing approaches possibly becoming a common vocabulary to which they can be referred and by which new approaches can be instantiated. Moreover, because of its abstract nature it can eventually become an instrument of models interchange between different approaches.

From the designed CDM we have then defined a domain specific modeling language (DSML) [17] for context awareness modeling we have called CAMEL (Context Awareness ModEling Languge). The essential feature of DMSLs is that specialized domain concepts are rendered as first-order language constructs, as opposed to being realized through a combination of one or more general constructs. This can greatly ease the designer/developer's task because it enables direct expression of problem-domain patterns and ideas. The CAMEL language consists of an extenstension of the Unified Modeling Languages (UML) with domain specific constructs for the modeling of context aware behaviours.

Taking inspiration from the Model Driven Development paradigm (MDD) [18], which is the discipline in software engineering that relies on models as first class entities, we have

then developed a modeling framework, based on the Eclipse platform, that realizes an editor for CAMEL models also making possible the application of model transformations aimed at producing:

- metrics or other measurments giving the designer feedbacks about his/her desigin choices;
- documentation which can be shared with the system's stakeholders;
- code which actually implements them;
- other artifacts.

In the 4th and 5th chapters of this thesis we describe the Model Driven Development paradigm, with a brief description of the Eclipse [19] and the Eclipse Modeling Framework (EMF) [20], the CAMEL language and editor themselves.

In the 6th chapter we finally introduce the code level counterpart of our approach which consists of a domain specific language called JCOOL (Java Context Oriented Language)[21] that is a Java extension explicitly designed for context oriented purposes. JCOOL represents the code level implementation our conceptual domain model. It has been presented in [21] for the first time but then consistently modified to improve its expressiveness. JCOOL has to be considered only one of the possible target platforms into which CAMEL models can be automatically transformed by means of suitable transformation processes. In fact, because of the platform independency of the CAMEL models, several transformations with respect to different target platforms are possible (i.e., AspectJ [22], Jasco [23], etc.).

In the 7th chapter we finally propose a simple application scenario aimed at providing a concrete example of how the developed framework can be exploited to introduce context awareness characteristics into a, possibly already existing, independently designed application. To this end the used scenario is the above described traveler service application introduced in [3]. Starting from a brief specification of this application, we depict how the Eclipse UML plugin can be exploited to produce a proper UML model of the base application which can be therefore referenced by our modelling framework to produce the models of the desired context aware features. We finally depict a possible JCOOL implementation of the specified CAMEL models that can be automatically generated by means of proper model transformations.

2 Understanding Context Awareness

Context is a long-standing concept in human-computer interaction. Currently existing interaction between humans and computers consist of explicit acts of communications (e.g. pointing to a menu item) and the context is implicit (e.g., default settings). In a near future context will be used to interpret explicit acts, making communication much more efficient. Thus by carefully embedding computing into the context of our lived activities, it can serve us with minimal effort on our part. Communication can be not only effortless, but also naturally fit in with our ongoing activities [24]. Pushing this further, the actions we'll take will be not even felt to be communication acts at all we'll rather engaged in normal activities and the computation will become invisible [25]. In a special issue of Human-Computer Interaction on Context in Design [26] dated 1994, there has been discussed the notion that the design of computing artifacts must take into account how people draw on and evolve social contexts to make the artifacts understandable, useful, and meaningful. Nowadays context-awareness is becoming a key function for consumer-oriented applications. Apple iPhone [27] for example is equipped with a set of accelerometers that detect when the user rotate the device vertically or horizontally, so that it can change the displayed contents in order to make the user see the entire width of a web page or a photo in its proper landscape aspect ratio. Because of its multi-touch display, that covers the entire device's surface, it is also equipped with proximity sensors that immediately turn off the display when the user lifts the device to its ear, in order to save power and prevent inadvertent touches until the device is moved away. Finally, ambient light sensors automatically adjust the display's brightness to the appropriate level for the current ambient light.

Recently, even video game consoles have introduced context aware capabilities that enrich the player's gaming experience. Nintendo Wii [28] for example distinguishes itself from the other consoles for its wireless controller, the Wii Remote, which can be used as a handled pointing device and detect movement in three dimensions. In fact a main feature of the Wii remote is its motion sensing capability, which allows the user to interact with and manipulate items on screen via movement and pointing through the use of accelerometer and optical sensor technology. The introduction of the Nintendo Wii provided millions of people with new ways to interact with computers and even researchers start to consider it an interesting platform to test new form of context-aware computers interaction [29].

The term “context-aware” has been introduced for the first time by Shilit and Theimer 1994 [30]. However, it is commonly agreed that the first research investigation of context-aware computing was the Olivetti Active Badge work in 1992 [31]. Since then, there have been numerous attempts to define context-aware computing. Today, the term “contex-aware computing” is commonly understood by those working in ubiquitous/pervasive computing, where it is felt that context is key in their efforts to disperse and mesh computing into our lives. However, even if the notion of context is much more widely appreciated today, there is still a lack of common agreement of what context awareness means and how it should be introduced in modern software systems.

2.1 Context and Context Awareness Definition

In order to use context effectively it is important to understand and agree on what context is and how it can be used. In fact, an understanding of context will enable application designers to choose what context to use in their applications, while an understanding of how context can be used will help application designers to determine what context-aware behaviors to support in their applications [32].

To develop a specific definition that can be used prescriptively in the context-aware software development field, we will look at how researchers have attempted to define context in their own work. While most people tacitly understand what context is, they find it hard to elucidate. At the beginning of the research toward context awareness, context was defined by examples. In [33,34,35,36] different examples of what context could be are provided, namely: the user’s location, identities of people and objects around the user, the user’s emotional state, date and time, etc.

Schilit et al. [37] define context to be the constantly changing execution environment. They include the following pieces of the environment:

- *Computing Environment*: available processors, devices accessible for user input and display, network capacity, connectivity and costs of computing, etc.;
- *User environment*: location, collection of nearby people, social situation, etc.;
- *Physical environment*: lighting and noise level, etc.;

Pascoe [38] defines context to be the subset of physical and conceptual states of interest to a particular entity. A generic definition of Context is provided by Dey et al. in [32] where the context is defined as “*any information that can be used to characterize the situation of an entity (i.e. wether a person, place or object) that are*

considered relevant to the interaction between a user and an application, including the user and the application themselves".

A further characterization of the context, is introduced by Kernchen et al. in [39] with the concept of user's sphere, which includes all the available devices (rendering components and interaction components) that are available to the user at a given time. This sphere is dynamic because the set of devices change over time depending on the context of the end user.

Together with a definition of what a Context is, several definitions of what are the characteristics of a context aware system have been provided. Dey et al. [32] propose an organization of context-awareness' definitions into two categories: *using context* and *adapting to context*. The first class encompasses all the definitions which consider context awareness as the ability of computing devices to detect and sense, interpret and respond to aspects of a user's context [40,41,42]; in the latter there are all those definitions which consider context awareness as the ability of an application to dynamically change or adapt its behavior based on the context of the application and the user [43,44]. Finally Dey *et al*. propose a definition of context-awareness which tries to encompass the previous definitions stating that "*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*".

2.2 Context Modeling

Context Oriented Modeling (COM) is the research area encompassing the all the approaches aimed at modeling context and contextual information. In [5] a survey about the most relevant context modeling techniques is provided. In this paper, Strang et al. propose a classification of the existing context modeling approaches into six categories which depend on the scheme of the data structures used to exchange contextual information in the respective system; these are: *key-value models*, *markup-scheme models, graphical models, object oriented models, logic based models, ontology based models.*

2.2.1 Key-value models

The first class, the *key-value models*, encompasses all those models adopting a key-value pairs data structure to represent context. Shilit et al [6] used key-value pairs to

model the context by providing the value of a context information to an application as an environment variable.

2.2.2 Markup scheme models

The second category consists of *markup scheme* modeling approaches which use a hierarchical data structure defined by markup tags with attributes and content, where the content of the markup tag is usually recursively defined by other markup tags. An example of this category is *the Pervasive Profile Description Language* (PPDL)[45], an XML-based language that allows to account for contextual information and dependencies when defining interaction patterns on limited scale.

2.2.3 Graphical models

The third category consists of *graphical models* such as UML. Several approaches have tried to address context modeling by suitable UML extensions [7,8,9,10]. In [9] it is proposed a context-extension of the Object-Role Modeling (ORM) approach whose basic modeling concept is the *fact*, and the modeling of a domain using ORM involves identifying appropriate *fact types* and the *roles* that entity types play in these. The proposed context based extension allows fact types to be categorized, according to their persistence and source, either as static (facts that remain unchanged as long as the entities they describe persist) or as dynamic. The latter ones are further distinguished depending on the source of the facts as either profiled, sensed, or derived types. Another indicator is a history fact type to cover a time-aspect of the context. The last extension is fact dependencies, which represent a special type of relationship between facts, where a change in one fact leads automatically to a change in another fact. In [10], the same authors propose an interesting number of observations about the nature of context information in pervasive computing systems by which they have determined the design requirements for their model of context. These are:

- C*ontext Information Exhibits a Range of Temporal Characteristics*: Context information can be characterized as static or dynamic. Often, pervasive computing applications are interested in more than the current state of the context. Accordingly, context histories (past and future) will frequently form part of the context description;
- *Context Information is Imperfect:* Information may be incorrect if it fails to reflect the true state of the world it models, inconsistent if it contains contradictory

information, or incomplete if some aspects of the context are not known. These problems may have their roots in several causes such as the fact that sensors, which are the context information producers, may provide faulty information;

- *Context Has Many Alternative Representations:* Much of the context information involved in pervasive systems is derived from sensors. There is usually a significant gap between sensor output and the level of information that is useful to applications, and this gap must be bridged by various kind of processing of context information;
- *Context Information is Highly Interrelated:* More or less obvious types of relationships may exist amongst context information. Context Information may be related by derivation rules which describe how information is obtained from one ore more other pieces of information.

In [7] Sheng et al. present a modeling language called ContextUML for the model driven development of context aware Web Services based on the Unified Modeling Language. ContextUML provides constructs aimed at generalizing context provisioning that includes context attributes specification and retrieval up to the formalizing of context awareness mechanisms and their usages in context aware web services. The language is introduced by a definition of its syntax including a metamodel and a notation. The former defines abstract syntax of the language while the notation defines the concrete format used to represent the language (also called concrete syntax).

2.2.4 Object oriented models

Object oriented models, try to employ the benefits of object orientation to address problems arising from the dynamics of the context in ubiquitous environments. In these approaches the details of context processing is encapsulated at an object level and hence hidden from other components. Access to contextual information is provided through specified interfaces only. The approach proposed in [46] can be taken as example of this context modeling category. In this paper, an architecture for sensor data fusion is proposed, where the information provided by physical and logical sensors can be accessed only by dedicate abstract entities called *cues*. A cue is regarded as a function taking the values of a single sensor up to a certain time as input and providing a symbolic or sub-symbolic output. A set of possible values for each cue is defined. Each cue is dependent on a single sensor but different cues may be based on the same sensors. The Context is then modeled as an abstraction level on top of the available

cues. Thus the cues are objects providing contextual information through their interfaces, hiding the details of determining the output values.

2.2.5 Logic based models

In *Logic Based* modeling approaches the context is defined as *facts*, *expression* and *rules* typically expressed in form of first order logic's predicates. In these approaches contextual information is usually added to, updated and deleted from a logic based system in terms of facts or inferred from the rules in the system respectively. In [47], one of the first logic based approaches for context modeling is proposed. In this paper, McCarthy et al. introduced contexts as abstract mathematical entities with properties useful in artificial intelligence.

2.2.6 Ontology based models

Ontology based context modeling approaches consist of those approaches which make use of ontologies to represent contextual information. Ontologies are a promising instrument to specify concepts and interrelations [48,49]. Some representatives of this category are the approaches proposed in [50,51,52].

In [50], Shehzad et al. propose an ontology driven approach to context modeling which is part of the CAMUS architecture, a middleware framework for context-aware ubiquitous computing [53]. In this framework a strong distinction is made between *context entities*, which are conceptual entities that can range from various kinds of devices to various environment conditions, and the information provided by them, called *contextual information*. Both these entities are defined by W3C's OWL (Web Ontology Language) [54] in domain ontologies which can be used by the system to share its knowledge between its entities and other systems and to apply logic inference in order to deduce new context information by the currently known.

Saidani et al. [51] have presented an approach for business processes modeling that support the description of the execution context, thus providing taxonomy of most common contextual information, and introducing context models (CM) which, they claim, allow to exhaustively structuring the contextual information in a convenient way. In this framework, the context is captured using *aspects*, which are non-functional features. Each aspect is addressed by some *facets*. Facets are described by attributes that are characterized by features that are directly measurable. Context Model is represented by a three level tree called Context Tree (CT) whose root represents the

global context, nodes at the first level refer to the aspects, nodes at the second level refer to facets and leaves refer to facets' attributes. The construction of the CT requires the competencies of the application domain expert which has to collect and structure the relevant context aspects, facets and attributes to define the appropriate functions allowing measuring them. At execution time the CT representation of a CM should be adapted to include only contextual information which is relevant to a BP. The Adapted Context Tree (ACT) will include only meaningful aspects, facets and attributes for the given BP. Finally, in order to instantiate BP using the ACT, an assignment activation strategy is introduced which means that only significant assignments, a sort of binding between entities of the BP and entities of the ACT, have to be taken into consideration in a given context. Hence, the set of assignments which mach better the current value of ACT is activated.

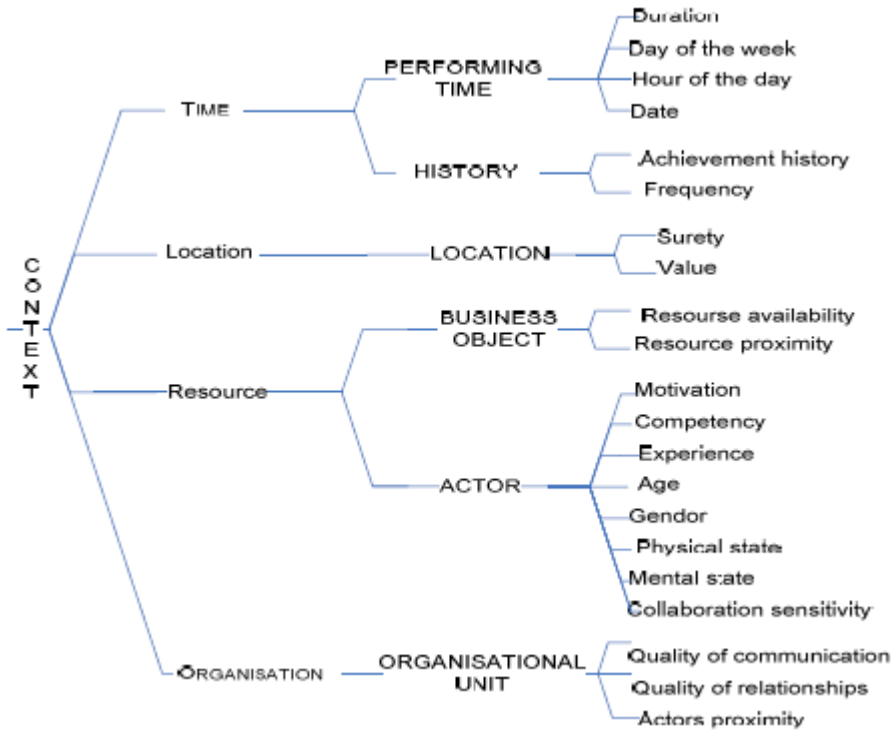

Figure 1: An example of Context Tree [51]

2.3 Context Programming

The research area that aims to address context aware issues at code level is called *Context Oriented Development* (COD) or *Context Oriented Programming* (COP) [11,12]. *Context Oriented Programming* is an emerging programming paradigm aimed at enabling the expression of structural and behavioral variation dependent on context. Similarly to

AOP techniques, the goal of Context-oriented Programming is to avoid having to spread context-dependent behavior throughout a program as a consequence of the inability of object oriented programming to provide proper constructs to encapsulate crosscutting concerns like context-awareness.

The term *Context Oriented Programming* has been used for the first time in two partially different meanings by Gassanenko in [13,14] and Keasy et al. in [15].

Gassanenko describes an approach to add object-oriented programming concepts to FORTH [55]. Furthemore a notion of context is introduced that allows code to behave differently when executed in different environments. In this approach Contexts are implemented as virtual method tables (VMT) that are not bound to the data and can be interchanged at runtime.

Keasy et al. [15] use the term *Context-oriented Programming* to define an approach that separates code skeletons from context-filling code stubs that complete the code skeleton to actually perform some behavior. The claimed advance is that the code stubs can vary depending on the context, for example the device some code runs on. A proof of concept implementation in Python and XML is described. Moreover Keays et al. introduced a number of requirements a COP languages should respect. Even though we think some of these requirements are probably too much related to the specific approach presented in [15] we have taken them as reference for the development of JCOOL, a COP language presented in the 6th chapter.

First of all, Keasy et al. state that in a COP language the Context must be a first-class construct. It therefore must form part of the lexical syntax of the language, having operators applied to it and being referenceable (for example stored in a variable).

In the Keasy's approach, context filling is the process where specified points in the target system are replaced by stubs (alternative chunk of code) depending on the context. The requirements for context filling are:

- **R1.1) Transparent**: The developer and user are unaware of the matching and binding process that is occurring during context-filling;
- **R1.2) Uses goal information**: Stub selections are made giving consideration to information available about the goals of the open term and stub;
- **R1.3) Uses context knowledge**: Stub selections are made giving consideration to information available about the context and the open term, the stub and the execution context;

- **R1.4) Uses inferred context knowledge**: A knowledge base should exist that allows inferences to be made about contexts;
- **R1.5) Selects closest match**: The stub whose goals and context most closely match those of the open term will always be selected;
- **R1.7) Scoping**: Where open terms are nested, those variables from each of the stubs that are declared 'bound', effectively bind into the same scope.

The requirements for context in COP, proposed in [15], are:

- **R2.1) First-class**: Context is a first-class construct;
- **R2.2) Context representation**: In its simplest form, context may be represented as attribute value, like in the key-value models [5]. However a COP language should have more elaborate features such as being able to express the contexts "the bandwidth varies by +/- 1Mbps" and "the students can see the screen";
- **R2.3) Operations**: Valid operations on context are disjunction, conjunction, subtraction, negation and comparisons;
- **R2.4) Inheritance**: A Context may be declared as an extension of an existing context, in which case only new or overridden contextual information needs to be declared;
- **R2.5) Comparable**: A context must be comparable to another on a continuous scale which represents how similar they are;
- **R2.6) Associations**: Context can be associated with open terms and stubs. When a context is associated with a stub it defines the contextual domain for that stub, i.e. what context that stub is valid for. When a context is associated with an open term it defines the context under consideration when a stube is to be found, or the contextual domain for that open term;

When dealing with pervasive environments the developer may not know the names of the available services and functions and cannot therefore refer to them by name. In the Keasy's approach *Goals* should be used to alleviate this problem. The requirements of goals are:

- **R3.1 First-class:** Goals are first-class constructs as defined above;
- **R3.2 Knowledge representation:** Goals need to have meaning to both humans and computers;
- **R3.3 Operations:** Valid operations on goals are disjunction, conjunction, subtraction and negation. Additionally, the composition operation merges the semantics of goals. For example the goal representing a credit to an account

could be composed with a goal representing a debit to another account to produce a third goal for transferring money between accounts;

- **R3.4 Inheritance:** A goal may be declared by extending an existing goal, adding or overriding information in it;
- **R3.5 Comparable:** A goal must be comparable to another on a continuous scale which represents how similar they are (not only 'equal' or 'not equal');
- **R3.6 Associations:** Goals can be associated with open terms, stubs, variables and parameters;

An Open term is a placeholder for a context-dependent section of a program. They are associated with a goal and a context and are replaced with a stub at runtime by the context-filling operation. Their requirements are:

- **R4.1) Second-class**: Open terms are second-class constructs, thus they are part of the lexical syntax of the language and may have operators applied to them but they may not be referenced;
- **R4.2) Is associated with a goal**: An open term is associated with a goal which provides information to assist selecting the stub that will replace it;
- **R4.3) Is associated with a context**: An open term is associated with a context which may what contextual information to consider during context filling and/or restrict the contextual domain that the open term is valid in, such that an open term may not be filled at all;
- **R4.4) May be associated with events**: When an open term is associated with an event the context-filling operation will only occur when this event occurs. Valid events are ente, exit, change and discover and refer to the context associated with an open term:
    - Enter: occurs when a context goes from being invalid to being valid;
    - Exit : occurs when a context goes from being valid to being invalid;
    - Change: occurs when any information in the execution context that is also in the valid context changes;
    - Discover: occurs when information from a context becomes a part of the execution context;
- **R4.5 Are ubiquitous**: An open term may be used in the place of any first or second-class programming construct;
- **R4.6 May be nested**: If an open term is filled by a stub which contains another open term, the second is filled in the same manner as the first;

- **R4.7 Operations:** Valid operations on goals are disjunction, substraction and negation. Additionally, the composition operation merges the semantics of goals;
- **R4.7 Inheritance:** Open terms may inherit goals and context from other open terms, adding to them or overriding them;

Stubs are pieces of code associated with a goal and a context that are used to replace open terms during the context-filling operation. They have the following requirements:

- **R5.1 Third-class:** Stubs are a third-class construct, as define above;
- **R5.2 Is associated with a goal**: A stub has a corresponding goal which provides information for the context-filling operation when an open term is filled;
- **R5.3 Is associated with a context:** A stub's context specifies its valid contextual domain;
- **R5.4 May use parameters**: a stub may contain references to variables that are not declared within the stub. Such variables may be bound to those in the scope of the host skeleton and are referred to as the stubs parameters;
- **R5.5 Parameters bound by goal:** A stubs parameters cannot be bound to a skeletons variables on a naming basis. This is due to the assumption that developers are not aware of the names of entities in a pervasive environment. Instead, parameters and variables must be associated with a goal and these are matched in the same way as open terms and stubs.;
- **R5.6 Bound, half-bound and unbound parameters:** Stubs must support three different types of parameters. Bound and half bound parameters get their value from the host skeleton, however only changes made to bound parameters retain these changes in the host skeleton's scope. Unbound parameters exist independently in the scope of the stub;
- **R5.7 Optional parameters:** Stubs should support parameters that are optional, either because a null value is acceptable or they have a default value;
- **R5.8 Default parameters:** Stub parameters may be given default values;
- **R5.9 Functions as stubs:** Normal functions can be used as stubs through the use of a wrapper.

A skeleton is simply a term given to any part of a program which contains open terms. Stubs, classes, functions, and expressions could all be considered skeletons providing they contained at least one open term. Skeletons have the following requirements:

- **R6.1 Context Free:** No section of code in a program skeleton may be context-dependant. A statement such as if (context1){ call method() } is illegal as it makes the call method() block context-dependant. Instead, the entire statement should be left as an open term, and call method() may be written as a stub associated with context1;
- **R6.2 Reuseability:** Program skeletons are usable across a number of different hardware and software platforms.

2.3.1 ContextToolkit

In [56] Salber et al. introduce a Java toolkit which relies on the concept of context widget. Context widget is a software component that provides applications with access to context information from their operating environment. In the same way GUI widgets insulate applications from some presentation concerns, context widgets insulate applications from context acquisition concerns thus providing the following benefits:

- They hide the complexity of the actual sensors used from the application;
- They abstract context information to suit the expected needs of applications;
- They provide reusable and customizable building blocks of context sensing.

From the applications' perspective, context widgets encapsulate context information and provide methods to access it in a way very similar to GUI toolkit.

Context widgets have a state and a behavior. The widget state consists of a set of attributes that can be queried by applications in order to retrieve information about the context perceived by the sensor associated to the widget. Applications can also register to be notified of context changes detected by the widget. The widget triggers callbacks to the application when changes in the environment are detected. Context widgets can then be considered as basic building blocks that manage sensing of a particular piece of context. The toolkit also provides means of composing widgets in order to make possible for the applications to take advantage of multiple types of context information. In this way the toolkit realizes a distributed architecture of widgets making possible the retrieving of context information from multiple distributed sources. The introduction of context widgets, with respect to an already existing application, should require the developers to modify the system's code in order to integrate sensors with the related widgets thus increasing the coupling with the context awareness and the business logic concerns. However the toolkit enables and speed up the development of context aware distributed applications by providing a technological platform with constructs making

possible an easy implementation of the context sensing and monitoring concerns. Contextual adaptation is not an objective of the toolkit which only makes possible to associate callbacks to the widgets. These callbacks represent the events that the widgets can use to notify contextual information to subscribing components and are automatically managed by the toolkit itself.

2.3.2 ContextJ

In [57] Hirschfeld, Costanza and Nierstrasz introduce a new COP approach identifying five essential properties a language should provide to support COP with respect to the context adaptation concern. These are:

- means to specify *behavioral variations*: variations tipically consist of new or modified behavior but may also comprise removed behavior;
- means to group variations into *layers*: layers group related context-dependent behavioral variations. Layers are first-class entities so that they can be explicitly referred to in the underlying programming model;
- dynamic activation and deactivation of layers based on *context*: Layers aggregating context-dependent behavioral variations can be activated and deactivated dynamically at runtime. Code can decide to enable or disable layers of aggregate behavioral variations based on the current context. Any information which is computationally accessible may form part of the context upon which behavioral variations depend;
- means to explicitly and dynamically control the *scope* of layers: the scope within which layers are activated or deactivated can be controlled explicitly. The same variations may be simultaneously active or not within different scopes of the same running application.

In ContextJ, Layers are first class constructs consisting of proper mechanisms of adaptation that can be activated in reaction to contextual information. Based on information available in the current execution context, specific layers may be activated or deactivated. The authors present their COP approach in the context of multi-dimensional message dispatch [57] not just because its implementation shall or shall not be based on such concepts but to help the reader to understand it as a continuation of other work, namely procedural, object oriented and subjective programming [58].
Message dispatch among the component of a system can be performed in one or more dimensional ways:

- *One dimensional dispatch* is characteristic of procedural programming. It provides only one dimension to associate a computational unit with a name. Here, procedure calls or names are directly mapped to procedure implementations;
- *Two dimensional dispatch* is characteristic of object oriented programming. It adds another dimension for name resolution to that of procedural programming. In addition to the method of procedure name, message dispatch takes the message receiver into consideration when looking up a method;
- *Three dimensional dispatch* is characteristic of subjective programming introduced in by Smith and Ungar. It extends object-oriented method dispatch by yet another dimension. Here methods are not only selected based on the name of message and its receiver but also on its sender;
- *Fourth dimensional dispatch:* introduced in [57] by Hirschfeld, Costanza and Nierstrasz, is characteristic of context oriented programming. It takes subjective programming one step further by dispatching not only on the name of a message, its sender and its receiver, but also on the context of the actual message send. Based on context information, methods or their partial definitions are selected for or excluded from message dispatch, leading to context dependent behavioral variations as described in the previous sections;

In Figure 3 there are two scenarios showing how context can affect method dispatch in COP environments. In both scenarios a message m1 is sent to the same receiver R*y*, from different senders (SA and SB) and in different contexts (Cα and Cβ). In Figure 3-a, the selection process results in method m1:*:Cβ. This is because that particular partial method implementation matches with messages m1, sent to receivers R*y* , by any sender (* is matched by sender S*A*), in both contexts Cα or Cβ. In Figure 3-b, the selection process results also in method m1:*:Cβ, now because the message and its sender and receiver correspond to the method's properties as before, and in addition to that context Cβ matches with m1's Cβ property as well. Method m1:*:Cα will not match with our request due to incompatible context properties.

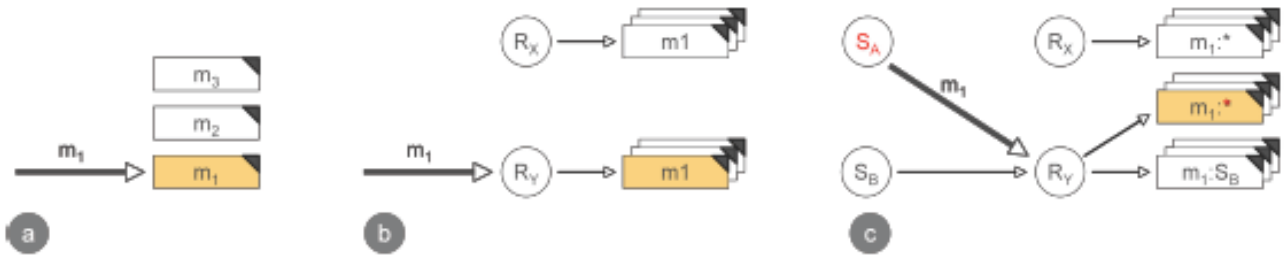

Figure 2: One-, two, and three-dimensional method dispatch [57]

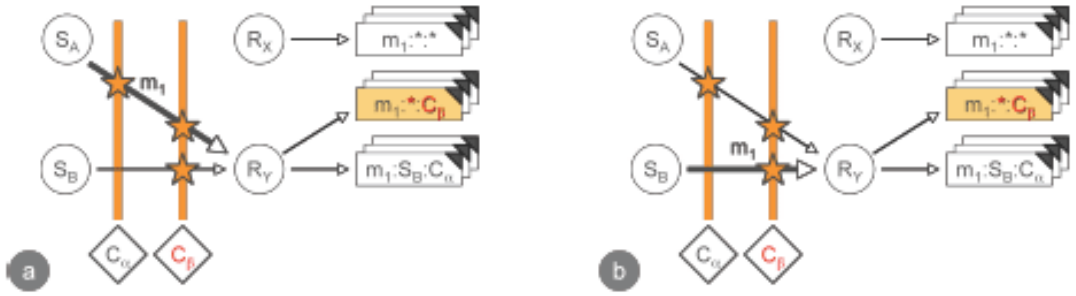

Figure 3: Four-dimensional method dispatch [57]

```
class Person {
  private String name, address;
  private Employer employer;

  Person(String newName,
         String newAddress,
         Employer newEmployer) {
    this.name = newName;
    this.employer = newEmployer;
    this.address = newAddress;
  }

  String toString() {return "Name: "+name;}

  layer Address {
    String toString() {
      return proceed()+"; Address: "+address;
    }
  }

  layer Employment {
    String toString() {
      return proceed()+"; [Employer] "+employer;
    }
  }
}
```

```
class Employer {
  private String name, address;

  Employer(String newName,
           Stringe newAddress) {
    this.name = newName;
    this.employer = newEmployer;
  }

  String toString() {return "Name: "+name;}

  layer Address {
    String toString() {
      return proceed()+"; Address: "+address;
    }
  }
}
```

Figure 4: An Example of ContextJ Layers [57]

The concepts introduced by Costanza et al. have been implemented in several context oriented extensions of the most important interpreted languages (*ContextP* for Python, *ContextR* for Ruby, *ContextJ* for Java, etc.). In this section we provide a brief description, throughout simple examples, of *ContextJ*.

In Figure 4 two java classes are defined which are respectively named *Person* and *Employer*. They have fields named *address* and *employer* together with the necessary constructors and a default *toString* method which in both cases return the value of the *name* attribute belonging to the related instance. ContextJ is then exploited to define two layers respectively named *Address* and *Employment* defining behavioural variations on the *toString* method. In the *Address* layer the address information is returned for instances of *Person* and *Employer* in addition to the default behaviour of *toString*. The *toString* method in the *Employment* layer instead, returns additional information about the employer of a person in the *Person* class.

None of the user-defined layers *Address* and *Employment* are activated by default. Instead, a client program must explicitly choose to activate them when desired. ContextJ provides *with* and *without* special constructs for the activation and deactivation of layers within dynamic scope.

The purpose of this simple example is to present different views of the same program where each client can decide to have access to just the name of a pearson, his/her employment status or his/her addresses, his/her employers or both. For example when a client chooses to activate the Address layer but not the Employment layer, address information of persons will be printed in addition to their names. When the Employment layer is activated on top, a request for displaying a person object will result in printing that person's name, its address, its employer and its employer's address, in that particular order. A code fragment showing the activation of these two layers is given in Figure 5.

```
Employer vub      = new Employer("VUB", "1050 Brussel");
Person somePerson = new Person("Pascal Costanza", "1000 Brussel", vub);

with (Address) {
  with (Employment) {
    System.out.println(somePerson);
  }
}

Output:  Name: Pascal Costanza; Address: 1000 Brussel;
         [Employer] Name: VUB; Address: 1050 Brussel
```

Figure 5: Example of ContextJ direct layer activation [57]

2.4 Context Driven Adaptation

The task of adding or changing system's functionalities at runtime depending of contextual information and without any side effects is challenging.
There exist several approaches to enable adaptation, mostly based on component models. They all may be categorized in three mayor approaches [59]:

- One possibility is the use of dynamic aspect oriented techniques or delegation models to apply modifications at runtime. Representatives of this technique are [60, 61, 62];
- Another approach is to use fractal components like [63], where components are compositions of subcomponents, which are target of adaptation by replacement.
- The third category adapts a component by replacement of its implementation, while the interface stays valid. [64, 65, 66, 67] proposed concepts realizing this approach.

The latter two approaches exhibit a large similarity in respect to the adaptation process. While the fractal model adapts a component by replacing small subcomponents behind a common interface, the last approach replaces the whole component behind its interface. Therefore the arising prerequisites, which have to be met to enable runtime adaptation, are the same, but in another granularity. For the sake of simplicity we call these two kinds of approaches *dynamic binding*.
In the following two sections we provide a description of both the aforementioned technological approaches, namely aspect orientation and dynamic binding, enabling the introduction of new behaviours into running software systems.

2.4.1 Aspect oriented approaches

To understand the concept of aspect orientation [68] we have first to look at the notion of concern and, consequently, at the separation of concerns principle (SoC)[69]. A concern can be defined as a property or area of interest of a system. Concerns can range from high-level notions like security and quality of service to low-level notions such as caching and buffering. They can be functional, like features or business rules, or nonfunctional (systemic), such as synchronization and transaction management. Because concerns refer to what the system has to do, they are generally directly or indirectly derived by system's requirements. The way concerns are distributed in the components of a system can actually be considered one of the key element of interest in software

engineering. In this area, the separation of concerns principle states that to achieve the required engineering quality factors such as robustness, adaptability, maintainability and reusability in the development of a system, each of its concerns should be mapped to one separated module. Otherwise, the system should be decomposed into modules so that each module realizes one concern were a module can be defined as an abstraction of a modular unit in a given design language (e.g. class or function). The advantage of this approach is that concerns are well localized and can therefore be easier understood, extended, reused, and adapted. Related with the SoC principle is the concept of cohesion. A module of a system is cohesive when it is designed to perform only a single precise task. Maximize the separation of concerns in a system means maximize the cohesion level of its components. An example of separation of concern, typical for software engineering, is the Model View Controller (MVC) design pattern [70]. This pattern, identifies three concerns involved in a generic data access application. These concerns are:

- the Model, the part of the application which represents the data to be accessed. It is the domain-specific representation of the information that the application operates.
- the Controller, the part of the application which manages and manipulates the data. It processes and responds to events, typically user actions, and may invoke changes on the model;
- the View, the part of the application which gives a certain representation of the data. It renders the model into a form suitable for the interaction, typically a user interface element. Multiple views can exist for a single model for different purpose;

A MVC control flow works as follow:

1. The user interacts with the user interface in some way (e.g., presses a button);
2. A controller handles the input event from the user interface, often via a registered handler or call back;
3. The controller accesses the model, possibly updating it in a way appropriate to the user's action;
4. A view uses the model to generate an appropriate user interface;
5. The user interface waits for further user interactions, wich begins the cycle anew.

Since the beginning of the modern software engineering there has been a continuous tendency to maximize the separation of concerns. In the first assembly languages there were not any construct to aid the developer in the separation of concerns. With the advent of procedural programming more abstract programming languages, such as C, introduced structures and procedures so that the developers started to think at software architectures in terms of functions. In the early ’90 Object Oriented Programming (OOP) consistently improved the instruments set of a software designer with new powerfull concepts like *Class*, *Object*, *Method*, *Inheritance*, *Encapsulation*, *Abstraction Polymorphism*. In OOP a program may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility. Because of the encapsulation principle OOP seemed to be adapt to address the separation of concerns principle at code level. Nowadays, object orientation is reflected in the entire spectrum of current software development methodologies and tools. Writing complex applications such as graphical user interfaces, operating systems and distributed applications while maintaining comprehensible source code has been made possible with OOP.

However, Even though object orientation is a clever idea it has certain limitations. Unfortunately, in the specification of a system, there are many requirements that can not be neatly decomposed into behavior centered on a single locus and the developer is forced to map the implementation of such concerns in many modules. This is called crosscutting while the concerns which cause a crosscut are called crosscutting concerns or Aspects [68].

Aspects are not the result of bad design but have more inherent reasons. A bad design including mixed concerns over the modules could be refactored to a neat design in which each module only addresses a single concern. However, if we are dealing with crosscutting concerns this principle is not possible, that is, each refactoring attempt will fail and the crosscutting will remain. A crosscutting concern is a serious problem, since it is harder to understand, reuse, extend, adapt and maintain it as its implementation will ressult spreaded over many places. Finding places where the crosscutting occurs is the first problem, adapting the concern appropriately is another problem. Since it is not easy to localize and separate crosscutting concerns, several modules will include more than one concern. We say that such concerns are tangled in the corresponding module.

Figure 6 depicts a graphical representation of possible concerns distributions with respect to a system's components. In this graph, circles represent the places where concerns crosscut a module. These are called join points. Join points can be present at the level of a module (class) or be more refined and deal with subparts of the module (e.g attribute or methods).

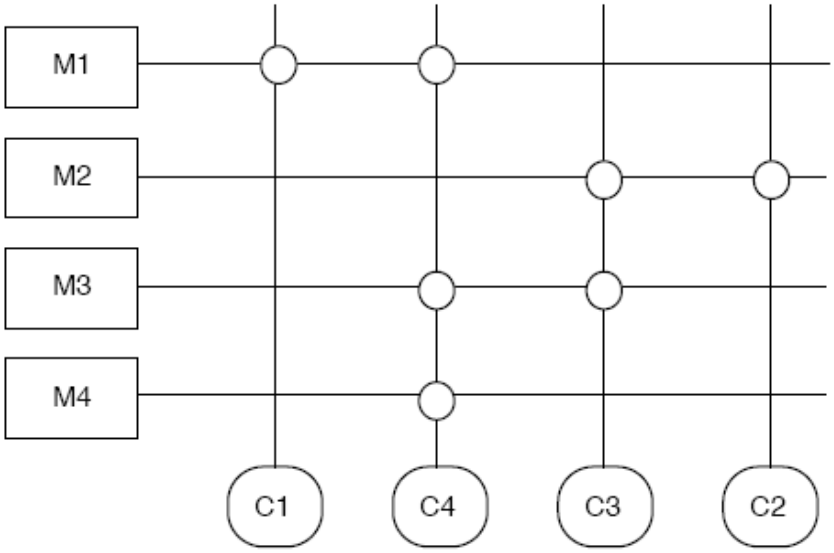

Figure 6: An example of concerns distribution into a system components

Crosscutting can be easily identified if we follow a concern in a vertical direction (multiple join points) tangling can instead be detected if we follow each module in the horizontal direction [71].

Several technological platforms already exist implementing aspect oriented approaches [72,22,23] sharing common characteristics. Any aspect oriented language typically provides a first-class construct, generally called Aspect, which encapsulates the logic of a single crosscutting concern. An aspect is generally defined in terms of where it crosscuts the modules of the base system and of what activities must be performed when those points are passed through by an execution flow. The constructs provided by an AO language to express where an aspect crosscuts the base system define its join point model (JPM). The JPM adopted by an Aspect Oriented Programming language dictates how the crosscutting modularization takes place [73].

Following the definition of Kiczales at al. in [68,72] a JPM is mainly composed of three elements:

- A set of points in the computational flow of a progran, called join point, that can be used to compose the separated concerns with the rest of the system;

- the pointcut definition language, that gives means of identifying the join points;
- a means of specifying semantics at join points (e.g. to execute code before, after or around a join point).

Join points are single atomic point in the execution of a program an aspect of an AO language is able to crosscut. Depending on the AO language, a JPM may be characterized by a more or less expressiveness set of join points.

Common join points are:

- Method call: When a method is called;
- Method execution: When the body of code for an actual method executes;
- Constructor call: When an object is built and that object's constructor is called;
- Constructor execution: When the body of code for an actual constructor executes, after its this or super constructor call. The object being constructed is the currently executing object, and so may be accessed with the this pointcut;
- Static initializer execution: When the static initializer for a class executes. No value is returned from a static initializer execution join point, so its return type is considered to be void.
- Object pre-initialization: Before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. Thus, the execution of these join points encompass the join points of the evaluation of the arguments of *this()* and *super()* constructor calls. No value is returned from an object pre initialization join point, so its return type is considered to be void.
- Object initialization: When the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object. The object being constructed is the currently executing object, and so may be accessed with the this pointcut. No value is returned from a constructor execution join point, so its return type is considered to be void.
- Field reference: When a non-constant field is referenced;
- Field set: When a field is assigned to. Field set join points are considered to have one argument, the value the field is being set to.
- Handler execution: When an exception handler executes. Handler execution join points are considered to have one argument, the exception being handled. No

value is returned from a field set join point, so its return type is considered to be void.

- Advice execution: When the body of code for a piece of advice executes.

The means of identifying join points is the pointcut mechanism. A pointcut is a predicate on join points, which is used to identify a set of join points. More generically, a pointcut is a set of join points that share common properties where one would link to control access rights so that we can consider a join point as an atomic pointcut.

Even though aspect orientation born as a code development technique it has recently started to be adopted for modeling purposes in order to improve the separation of concerns even at the modeling level. Several proposals exist [3] which goes from extending already existing modeling languages such, as the UML [74], to the definition of new modeling languages with aspect oriented capabilities.

Context awareness is itself a crosscutting concern. In fact both contextual information related to a context definition and the adaptation behaviors which are needed to react to an experimented context may affect several physical or logical sensors and system components. As a consequence researchers of context aware systems have started to look at aspect oriented technology in order to improve the way contextual information is retrieved and adaptation behaviors are introduced by/into such systems.

Several research works already exist related to this topic [75,76,3]. In [75], Tanter et al., introduce a general analysis of the issues associated with context-aware aspects thus providing a set of best practices for context-aware aspects design. Some of them are:

- *Contexts and context-aware aspects must be separated entities*: separating aspects that are dependent on some contexts from the definition of the contexts themselves serves well-established engineering principles such as separation of concerns (SoC)[69];
- *Contexts should possibly be stateful, composable and parametrized*: a context may have a state associated to it, otherwise it must be stateful; it may be defined from more primitive contexts, thus composable and may be defined generically and then parametrized by aspects that are restricted to it;
- *Contexts state should possibly be bound to pointcut variable*: Restricting an aspect to a particular context requires the possibility to refer to a context definition in a pointcut definition. This is semantically equivalent to restrict an aspect based on whether a certain context condition is currently verified;

- *It should be possible to express dependencies on past contexts*: finally, it should also be possible to restrict an aspect based on whether the application *was* in a certain context previously.

In [4] Grassi et al., introduce a UML profile for aspect oriented modeling which has been later extended [3], by the same authors, with stereotypes and constructs aimed at modeling context awareness. This profile can be considered a first, prototypical, UML instantiation of the conceptual model described in the 3rd chapter.

2.4.2 Dynamic Binding

When referring to a software system, being it stand alone or distributed, a behavior consists of a behaviour description (i.e. a method signature, a service description, etc.) and a corresponding behaviour implementation. Updating a behavior at runtime means to switch the behavior implementation (which is typically encapsulated in a component), while the behavior interface as part of the behavior description stays valid. This implies that the new version of the implementation has to fulfil the same behavior interface. To enable this capability at runtime, the following requirements introduced in [77] have to be accomplished:

- *Transparency*: The replacement of a behavior implementation has to be transparent for the depending behavior and applications. No extra code should be included into the implementation of a behavior to handle the replacement of its dependencies. But it is justifiable that a component implements special methods to consider the replacement of itself;
- *Atomicity*: Changing a behavior implementation must be uninterrupted and therefore an atomic operation. Other behaviors or application are not allowed to see any intermediate states. Accessing an intermediate state may also result in unpredictable side effects;
- *State preservation*: Attributes of the affected component may have a specific value at the moment of change. This state of the behavior has to be transferred to the new implementation. Therefore the state of the old version of the underlying component has to be saved and injected into the new version;
- *Lifecycle management*: The whole adaptation process has to be coordinated by the environment. A Lifecycle management has to control the process: Saving the state of the old version, replacing the old version by the new one and restoring

the state. The Lifecycle management must also ensure the atomic execution of the whole process.

A number of research works already exist proposing technological approaches to enable dynamic bindings [64, 65, 66, 67]. In [65] Mukhija et al. investigate the problem arising from dynamic recomposistion of a target system's components introducing CASA (Contract-based Adaptive Software Architecture) which is a framework enabling dynamic adaptation of applications executing in dynamic environments. In this paper a component composition is defined as a collection of components qualified to do the required application task under a specific state of the execution environment. An adaptive application needs to provide a number of alternative compositions for different states of the execution environment. As a consequence, dynamic recomposition of components consists of changing between alternative compositions of an application at runtime. When changing from one alternative composition to another, there may be some new components to be added and some old components to be removed. Dynamic replacement of components is a special case of a dynamical removal of a component *A* followed by a dynamic addition of a component *A'*, such that *A'* is able to serve all those components that could be served by *A*, in an alike manner as *A* itself. In [67] Bidan et al. proposes an approach which tries to address the issue of dynamic reconfiguration in the CORBA framework also definding consistency and efficency constraints that have to be respected during the reconfiguration.

3 A Conceptual Domain Model for Context Awareness

Starting from the different works summarized in the chapter above, we have defined a conceptual domain model (CDM) which tries to encompass them all thus providing a common baseline by which new approaches can be instantiated. A conceptual domain model is a model representing concepts and their relationships aimed at providing a definition of those elements which are needed to describe concerns related to a specific domain, in our case the context awareness.

Our CDM, already presented in [3] and here properly refined, is quite close to the work presented in [7]. With respect to it we propose a more extended context model (including for example histories of events) and a more refined definition of context awareness mechanisms. We consider the context awareness concern as consisting of three distinct parts: *context sensing*, *context adaptation triggering* and *context adaptation*.

Context sensing encompasses the set of activities aimed at retrieving *contextual information* from physical or logical sensors.

Context adaptation triggering is defined by the set of those activities that continuously evaluate sensed, or possibly inferred, contextual information and, depending on certain rules, trigger the execution of adaptation mechanisms.

Context adaptation is finally defined by the set of adaptation mechanisms that can be triggered and then activated in response to context adaptation triggering activities thus reacting to context changes.

In our conceptual model these three concerns are represented by concepts contained into three different packages namely: *ContextData (COD)*, *ContextAdaptationTriggering (CAT)* and *ContextAdaptation (COA)*.

The semantic we give to CDM packages is similar to the semantic of MOF packages [78] as they act as container for logically related concepts. Figure 7 depicts the relationships among the packages which realize our conceptual model. Also the semantic of package relationships is similar to the one defined for the MOF packages [78], namely:

- Generalization (<<extend>>): When one package inherits from another, the inheriting sub-package acquires all of the metamodel elements belonging to the super-package it inherits from. Package inheritance is subject to rules that prevent name collision between inherited and locally defined metamodel elements;

- Importation (<<import>>): when one package imports another, the importing package is allowed to make use of elements defined in the imported package. However unlike the subtype package an importing package does not have the capability to create instances of imported Classes;

Both the *ContextAdaptation* and the *ContextAdaptationTriggering* packages import the *ContextData* so that they can refer to the concepts it contains. The *ContextAdaptation* package also imports the *ContextAdaptationTriggering* because the concepts contained in the first one need to refer to some concepts contained in the latter.

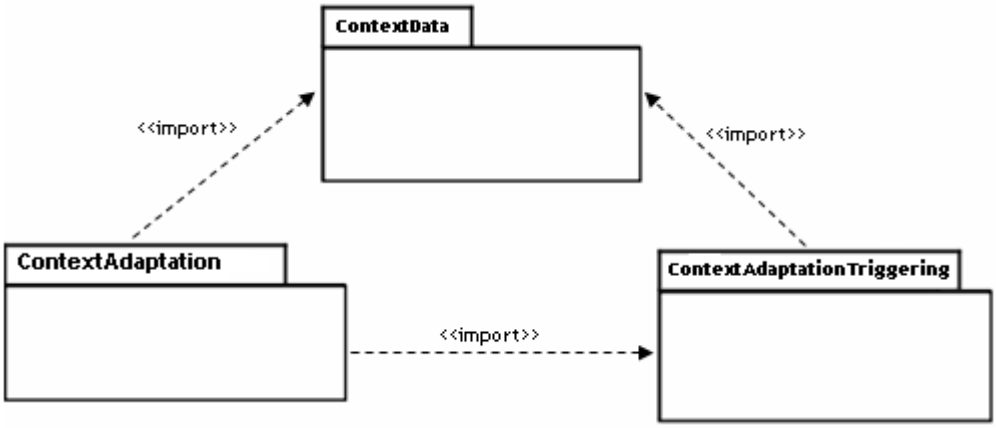

Figure 7: Conceptual Model Package Structure

3.1 Context data Package

The *ContextData* package contains all those concepts that are useful to model context sources and contextual information. All these concepts, together with their attributes and relationships, are described in this paragraph and briefly summarized in the table depicted in Figure 10.

Following the aforementioned definition of Dey et al. [32] in our conceptual model a context can be defined as "*any information that can be used to characterize the situation of an entity (i.e. wether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves*". As proposed by Shehzad et al In [50], we call such an entity a *context entity*. To model context entities a generic *Entity* concept is defined. It can be used to represent all those entities the context model elements need to refer to, either external or internal to the context models themselves.

As in the framework proposed by Shehzad et al. we call *contextual information* (or *context information*) that information, related to one or more context entities, that has to be retrieved in order to define a context. This information can have different form, depending on the way it is sensed and perceived by the system and strictly relates to the kind of context entities it refers to. As in the *key-value model* [5] contextual information can be represented as a set key-value pairs that define the current state of a given context entity (its temperature, its location, its energy, etc.) or it can be more natural to express it as a set of events directly or indirectly related to the context entity, that are supposed to be relevant to define its context [9]. These approaches are surely isomorphic. The state of any entity can in fact be both defined as the set of value assumed by its constituent characteristics, or by the set of events, either internal or external, which has led from a starting state to the current one. However, depending on the kind of contextual information to be handled and on the available ways by which it can be retrieved, the designer may prefer to adopt one approach respect to the other. He/She may also prefer to adopt an hybrid approach that makes him/her possible to define a context as a set of elements taking into account both the dynamical (events) and the statical (key/value pairs) characteristics of an observed system. It is reasonable, as an example, that a designer should not be interested in knowing all the possible sequence of events which led a device such as a mobile phone in a state of few available energy. Therefore he/she may probably find more natural to model this kind of information by exploiting a key-value pair approach, for example by a boolean pair which is true whenever the mobile phone is close to finish its available energy, independently of the way it is entered in this condition. At the same time the designer may be interested in capturing all those conditions which can be energy intensive such as the use of a camera, the activation of a Bluetooth connection, etc., which are information that can be easily handled as events. Furthemore, the designer can be interested in knowing whenever an energy intensive event occurs while the mobile phone is in a state of few available energy; which is a typical scenario he/she may find natural to model with an hybrid (key-value and events) approach.

Because of what argued we have introduced a set of concepts representing both these two approaches. The *StateBasedContext* and *EventBasedContext* (Figure 8) concepts respectively represent the key-value and the event based way to model contexts.

The former concept consists of a set of attributes, represented by the *ContextAttribute* concept, that are supposed to be relevant for a given context entity. A context attribute

is characterized by a name, which is unique in the context it belongs to, and a value. A state based context can be therefore considered as a container for logically related key-value pairs.

For example, in order to model the context of the available resources of a mobile phone, a state based context can be modeled in which a context attribute is defined containing the value of the still available energy percentage.

This attribute can then be associated by means of a *source* relation to a context entity that will provide values for it. This entity can be any available resource (either physical or logical) that can be exploited to retrieve the needed information. As an example, it can be a physical address in the mobile phone memory storing this information.

An event based context instead, consists of a set of events, represented by the *ContextEvent* concept, that are supposed to be relevant for a context entity. A context event can refer to any event which can be sensed and perceived by the system. As an example, in order to model the context of a mobile phone's user activities, an event based context can be modeled in which a subset of context events can be defined representing those events that are typically energy intensive (the activation of the camera, etc.).

Context attributes and context events are therefore conceptually similar to the pointcut concept used in AOP. They in fact represent points of a target system where contextual information has to be spilled out as pointcuts represent points in a target system where code implementing a concern has to be introduced.

A *Context* can finally be built as an aggregate of other contexts, both state-based and event-based by means of a *CompositeContext* (Figure 8). As an example, because the information about the available energy and about the occurrence of energy intensive events are logically related, the designer may decide to bring them together by defining a composite context containing both the state based and the event based contexts we have previously defined.

Temporal information also has to be considered as first class element in the modeling of a context. In fact, as depicted in [10, 75], contextual information can be characterized as static and dynamic. Static contextual information describes those aspects of a pervasive system that are invariant such as a person's date birth. However, as pervasive systems are typically characterized by frequent changes, the majority of information is dynamic. The persistence of dynamic contextual information can be highly variable; for example relationships between collegues typically endure for months or years, while a

person's location and activity often change from one minute to the next. The persistence characteristics influence the means by which contextual information must be gathered. While it is reasonable to obtain largely static context direct form users, frequently changing context must be obtained in indirect means, such as through sensors. Often, pervasive computing applications are interested in more than the current state of the context. As an example, consider an application enabling the user to research facilities of interest (such as restaurants, cinema, etc.) which are currently close to him/her. Such application can be enhanced in order to exploit old user's selections with the aim to suggest him/her facilities ordered with respect to his/her inferred preferences.Accordingly it should be possible to restrict the interest on a context information on whether the application was in a certain context previously observed

In our conceptual model a given time instant is represented by the *TimeSpecification* concept. This abstract concept is realized by the concrete concepts *ClockTime* and *EventTime*. The latter identifies the time instant at which a *ContextEvent* has occurred while the former identifies, by a precise clock specification, a time instant in which a Context held. Such difference can be usefull when the designer's intention is to retrieve contextual information associated to a possibly past event occurrence (i.e. the user's has done a certain operation in the execution history, etc.) or when the need is to retrieve contextual information associated to a precise time (i.e. the user agend for the next week, etc.). Therefore such approach makes the *TimeSpecification* concept exploitable in order to refer to past contexts either event or state based.

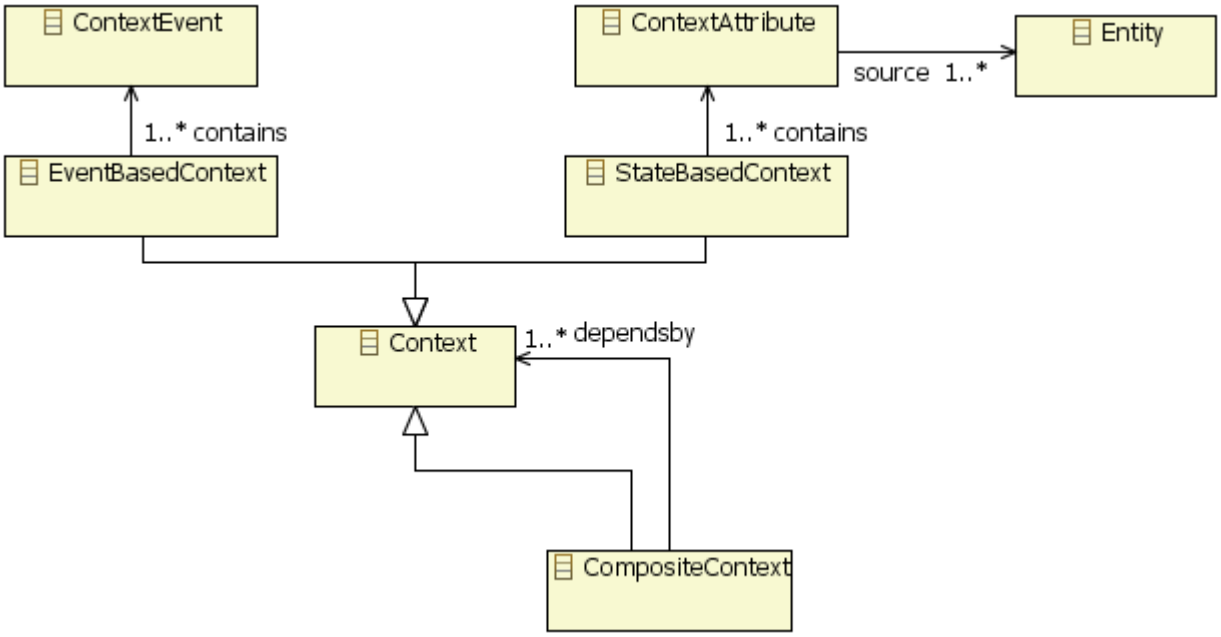

Figure 8: Context Conceptual Model

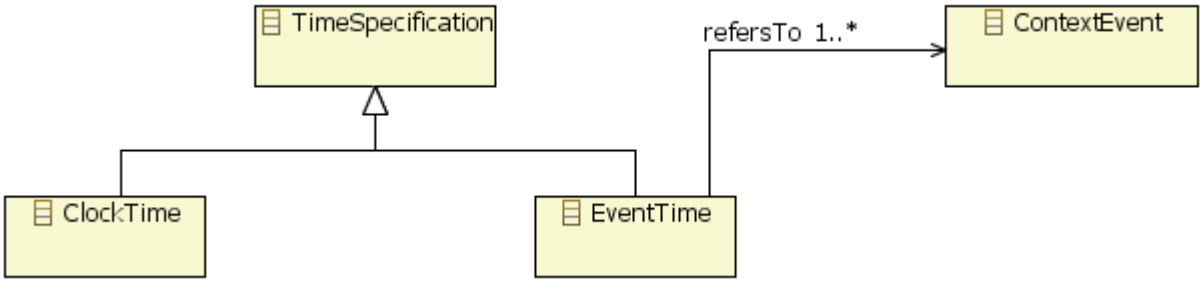

Figure 9: The Time Specification concept

Concept name	**Semantics**	
Entity	It is the concept representing an entity involved in the context awareness model. This entity can be either an external or internal to the context models themselves	
	Attributes & Associations	**Semantics**
	Name	Each entity is characterized by a name which uniquely identifies it
Concept name	**Semantics**	
Context	Abstract concept representing logically related context information retrieved from one or more context entities	
	Attributes & Associations	**Semantics**
	Name	Each context is characterized by a name which uniquely identifies it
Concept name	**Semantics**	
CompositeContext	Concept representing complex contexts whose definition partially or totally depends on the information provided by other context	
	Attributes & Associations	**Semantics**

	Name	Inherited from the *Context* concept
	dependsBy	Association which relates a composite context with another context by which it depends to define composite context information
Concept name	**Semantics**	
StateBasedContext	Concept representing a special kind of context modeled as a set of key-value pairs represented on the *ContextAttribute* concept.	
	Attributes & Associations	**Semantics**
	Name	Inherited by the *Context* concept
	Contains	Aggregation which relates a state based context with the context attributes it contains.
Concept name	**Semantics**	
EventBasedContext	Concept representing a special kind of context modeled as a set of events represented by the *ContextEvent* concept.	
	Attributes & Associations	**Semantics**
	Name	Inherited by the *Context* concept
	Contains	Aggregation which relates an event based context with the context events it contains.
Concept name	**Semantics**	
ContextAttribute	Concept consisting of a key-value pair representing a contextual information retrieved from a context entity.	
	Attributes & Associations	**Semantics**
	Name	Each context attribute is identified by a name which is unique with respect to the state based context containing it.
	Source	Association which specifies from which entity the context attribute takes its value
Concept name	**Semantics**	
ContextEvent	Concept representing an event which is relevant for the definition of a context.	
	Attributes & Associations	**Semantics**
	Name	Each context event is identified by a name which is unique with respect to the event based context containing it.
Concept name	**Semantics**	
TimeSpecification	Abstract concept representing a time instant.	
Concept name	**Semantics**	
ClockTime	Concept representing a precise time instant by means of a clock time specification.	
Concept name	**Semantics**	

EventTime	Concept representing a precise time instant as the instance of a context event occurrence	
	Attributes & Associations	**Semantics**
	refersTo	Specifies the context event to which the event time relates

Figure 10: the Context Data package concepts

3.2 Conceptual Modeling of Context Adaptation Triggering

Context adaptation triggering consists of those set of activities that monitor the context of context entities evaluating their contextual information and possibly triggering the introduction of adaptation mechanisms when certain conditions are verified. In our conceptual model, the concepts needed to model these entities are contained in the *Context Adaptation Triggering (CAT)* package. These concepts have to be considered as conditions that must hold to trigger the activation of proper adaptation mechanisms in order to react to the currently observed contexts. All these concepts, together with their attributes and relationships, are described in this paragraph and briefly summarized in the table depicted in Figure 13. In our conceptual model the trigger which causes the introduction of adaptation mechanisms is represented by an abstract concept named *AdaptationTrigger*. The *AdaptationTrigger* concept is modeled as an extension of the *ContextEvent* concept. This means the activation of an adaptation mechanism can be considered itself as an event belonging to the definition of a context.
The condition that states if an adaptation trigger is activated is represented by the *ContextConstraint* abstrat concept to which the *AdaptationTrigger* concept relates by a *constrainedBy* association. Two different types of context constraints are possible represented by the *StateConstraint* and *EventConstraint* concrete concepts (Figure 11), which respectively refer to the two aforementioned approaches to model context, *StateBasedContext* and *EventBasedContext*.
A state constraint is defined by a logical predicate over the value of the attributes of a state-based context. Hence, this constraint holds at some time instant, if the value of the involved context attributes at that instant satisfies the constraint condition (where the context is idenfitied through the *uses* association). Consider as an example a state based context with a context attribute representing the available energy of a certain device, such a mobile phone. A state constraint can be defined that is verified whenever the value of such context attribute is lesser than a specified threshold in order to capture a state of few energy.

An event constraint is instead defined as a precise pattern of events. The events used to define a pattern for an event constraint can be selected among the previously defined context events, including other adaptation triggers. This constraint holds at some time instant if the specified pattern can be identified within the history of context events that have occurred up to that instant in the associated event-based context. The task of modeling pattern of events can be considered similar to the identification of valid sentences in the theory of formal languages [79], where the set of possible patterns can be considered as the set of valid sentences with respect to a vocabulary of events and a grammar which can generate these patterns. In [79], Noam Chomsky, defined a hierarchy of possible grammars characterized by an increasing level of expressiveness (Figure 12). The lesser expressive grammar can generate regular languages, which can be represented by finite state automata, while the most expressive can generate *recursively enumerable* languages, which only a *Turing machine* can handle. Depending on the needs of the specific application in which our conceptual domain model has to be instantiated, a proper grammar to represent pattern of events has to be chosen. We therefore do not further specify how these patterns can be defined giving to the designer the possibility to choose the kind of model that is more suitable for his/her purposes (state machines, Petri-net, push-down automata, etc.).

As an example, given an event based contexts which collects context events representing those events that are energy intensive for a mobile phone (the activation of the camera, the activation of the Bluetooth connection, etc.). An event constraint can be defined as a simple pattern consisting of an OR of these events so that it instantaneously holds whenever an energy intensive event occurs.

Both the *StateConstraint* and *EventConstraint* concepts extend the *ContextConstraint* abstract concept which is characterized by a relation of composition with itself so that a context constraint may also be built by a composition of other constraints both state and event based. The composition of constraints can be done by means of proper operators that depend on the kind of constraint to which they are applied. Both state constraints and event constraints can be composed by logical operators (AND, OR, NOT, etc.). Arithmetical operators (<,>,=,etc.) can be used to compose state constraints while event constraints can be composed in pattern defined by means of a proper Chomsky grammar. As an example the aforementioned state constraint and event constraint can be composed together with an *AND* operator so that the obtained composite constraint holds whenever an energy intensive event occurs while the mobile phone has few

available energy. Context constraints represent the building blocks by which context adaptation triggers can be defined. They define conditions over the contextual information brought by context-entities that, if verified, identify a context of interest. In other words context constraints reduce the possibly infinite set of values context-entities may assume, by means of their contextual information (context attributes and context events), to a subset of interest. This subset consists of contextual states represented by the *ContextState* concept which is defined as a concrete realization of the *AdaptationTrigger* abstract concept. Because context states are themselves adaptation triggers and thus context events, their occurrence can also be part of event constraints for the occurrence of other context states. Logically related context states are finally grouped together by entities called monitors represented by the *Monitor* concept. Following the mobile phone example, a monitor containing context states related to the system energy management can be defined. It may for example consist of a context state representing the condition in which the device has few energy available and an energy intensive operation is requested by the user (the activation of the camera, etc.). This context state should relate to the composite constraint defined above which brings together by an *AND* operator the state constraint capturing the few energy available context information and the event constraint capturing the requested energy intensive operation information.

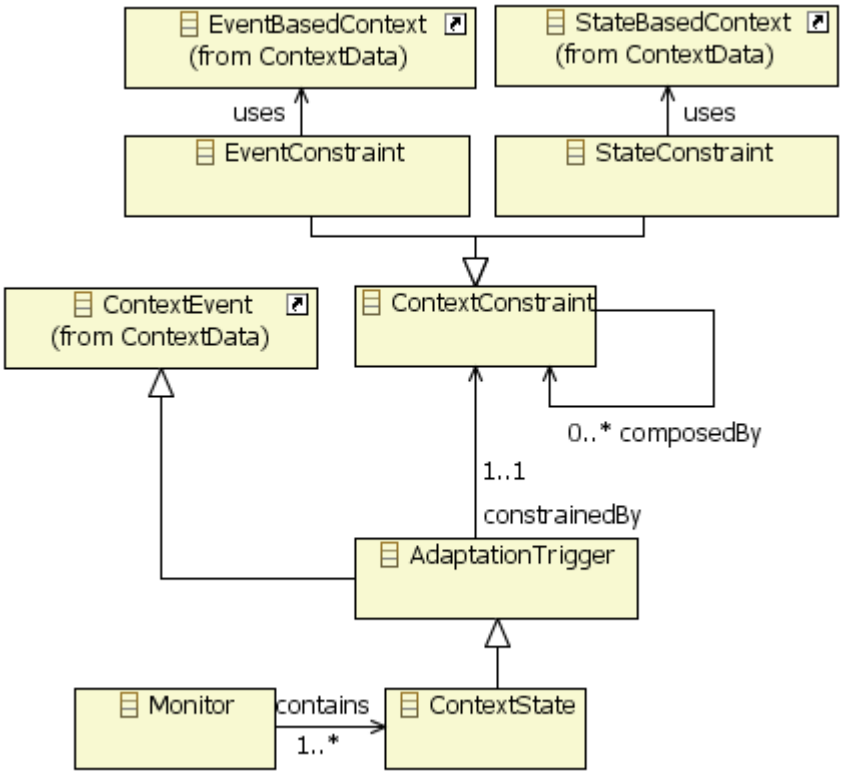

Figure 11: Context Adaptation Triggering Package

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

Figure 12: The Chomsky hierarchy

Concept name	Semantics	
AdaptationTrigger	Concept representing an element which triggers the introduction of adaptation mechanisms in order to react to context change	
	Attributes & Associations	**Semantics**

	name	Inherithed from the *ContextEvent* concept
	active	Boolean attribute which is true when the *AdaptationTrigger* is active, false otherwise.
	constrainedBy	Associate an adaptation trigger to the concext state representing its activation condition
Concept name	**Semantics**	
ContextConstraint	Abstract concept representing a condition over a given set of context-information.	
	Attributes & Associations	**Semantics**
	name	Each context constraint is characterized by a name which uniquely identifies it.
	composedBy	The activation of a context constraint can be defined by means of a function of other context constraints through the *composedBy* association. Such function is appositely not well defined in this conceptual domain so that it can be realized in several possibly different ways. As an example it might be implemented by means of proper operators that depend on the kind of constraint to which they are applied. Both state constraints and event constraints can be composed by logical operators (AND, OR, NOT, etc.). Arithmetical operators (<,>,=,etc.) can be used to compose state constraints while event constraints can be composed in pattern defined by means of a proper Chomsky grammar.
Concept name	**Semantics**	
EventConstraint	It is a special kind of context constraint which is defined by a precise pattern of context events and adaptation triggers. This constraint is verified if the specified pattern can be identified within the history of events that have occurred up to that instant in the associated event-based context.	
	Attributes & Associations	**Semantics**
	name	Inherited from the *ContextConstraint* concept
	composedBy	Inherited from the *ContextConstraint* concept
	eventPattern	The pattern of events which defines the event constraint condition. This pattern can be composed by context events and other adaptation triggers.

	uses	Association which relates the event constraint with the event based contexts owning the context events used to define the *eventPattern*.
Concept name	**Semantics**	
StateConstraint	It is a special kind of context constraint which is defined by a logical predicate on the value of the attributes of a state-based context	
	Attributes & Associations	**Semantics**
	name	Inherited from the *ContextConstraint* concept
	composedBy	Inherited from the *ContextConstraint* concept
	activationCondition	Logical predicate on the value of the attributes of state-based contexts
	uses	Association which relates the state constraint with the state based contexts owning the context attributes used to define the activation condition.
Concept name	**Semantics**	
ContextState	It is a special kind of adaptation trigger which identifies a state of interest with respect to a given context. It differs from *ContextConstraint* because it is not a condition but a state in which a given context shall or shall not be. The condition that states if a context state is active is given by the *composedBy* association with a context constraint. The context state is active (inherited active attribute equals to true) when the context constraint to which it is associated holds while it is not active otherwise.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the *AdaptationTrigger* concept
	Active	Inherited from the *AdaptationTrigger* concept
	constrainedBy	Associate a context state to the context constraint representing its activation condition
Concept name	**Semantics**	
Monitor	It is a container for logically related context states.	
	Attributes & Associations	**Semantics**
	Name	Each monitor is identified by a unique name
	Contains	Aggregation which relates a Monitor with the context states it contains.

Figure 13: The Context Adaptation Triggering concepts

3.3 Conceptual Modeling of Context Aware Adaptation

Context Aware Adaptation can be defined as the set of adaptation mechanisms that can be triggered and then activated during context monitoring activities in order to properly react to context changes.

The *Context Adaptation* package (COA) consists of a set of concepts that can be exploited to model adaptation mechanisms. In this package, a generic adaptation mechanism is represented by the *AdaptationMechanism* abstract concept.

We have identified two fundamental mechanisms to introduce context-awareness into an application:

- by *context-aware bindings*: representing the adaptation approach of dynamic binding (section 2.4.2). A binding associates values to application entities, depending on the retrieved contextual information. In our conceptual model a binding is represented by the *Binding* concept which extends the *AdaptationMechanism* abstract concept.
- by *context-aware inserts*: representing the aspect oriented adaptation approach (section 2.4.1) which introduce additional structural elements or behaviors at specific "points" of the application, depending on the context. In our conceptual model an insert is represented by the *Insert* concept which extends the *AdaptationMechanism* abstract concept.

A binding is defined by a pair consisting of an entity and a set of one or more values. When a binding is activated it provides a value for the specified target entity which substitutes the previous one. Depending on the kind of the entity this mechanism is applied to, it can be used to achieve different types of adaptation. For example, it can be used to bind a service interface to different implementations, a service invocation to different services, a parameter in a service invocation to different values. The value bound to a context entity may depend on its context so that it has to be evaluated by a *BindingOperation*. The binding operation is a concept representing the rule that prescribes how the binding may depend on context information. It is therefore characterized by a relation with the *Context* concept and by an *evaluatedAt* association with the *TimeSpecification* concept. The *evaluatedAt* assoaciation can be exploited in order to constraint a binding operation even to past contexts.

The concept of *context-aware insert* is derived from the domain of AOSD and AOP [68] (section 2.4.1), it is represented by the *Insert* abstract concept and models the introduction of new structural or behavioral elements into an already existing entity. It

therefore consists of a specification of the value that must be introduced, and of the point within some application entity where it has to be introduced. We distinguish two types of *Inserts*: a *StructuralInsert* and a *BehavioralInsert* both extending the *Insert* abstract concept. In both these cases, they consist of a specification of the value that must be inserted, and of the point within some application entity where it must be inserted. More than one value can be associated with a given insert. In this case, to select the value to be inserted, each value in the specified set is associated with a context constraint through a *selectedBy* association. A structural insert basically corresponds to the concept of *intertype declaration* used in Aspect Oriented Programming. Each structural insert is therefore associated with a *StructuralPointcut*, which specifies the part of the static structure of the application which is affected by the mechanism, and a *StructuralValue*, which is used to specify the structural elements that must be introduced. This mechanism could be used, for example, to introduce additional parameters within a service invocation or new components that provide additional functionalities.

A behavioural insert basically corresponds to the concept of *advice* used in AOP. It is associated to a *BehavioralPointcut* which corresponds to the join point concept of AOP and is used to specify where, in the application dynamics, an additional behavior must be introduced, and to a *BehavioralValue* which is used to specify the additional behavior itself. The *AdaptationLayer* concept is derived from the context oriented programming approach of Costanza et al. [57] and consists of another concrete realization of the *AdaptationMechanism* abstract concept. As in the work of Costanza an adaptation layer represents a collection of logically related adaptation mechanisms, such as bindings or inserts. When an adaptation layer is activated all its contained adaptation mechanisms are activated too. As a consequence the deactivation of an adaptation layer causes the deactivation of all the adaptation mechanisms it contains.

Logically related adaptation layers are contained by *Adapters*. An adapter can be therefore imagined as an entity which receives signals by one or more logically related monitors whenever one of their context states goes active. When a Monitor detects the activation of a context state it notifies the event to the interested adapters which causes the activation of the related adaptation layers that respectively cause the activation of their adaptation mechanisms. An adaptation layer has to be considered active until the related context state holds; whenever an adaptation layer is no longer active, its adaptation mechanisms are removed. Figure 14 depicts a graphical

representation of the conceptual architecture described above. The left side of such figure depicts the architecture of an hypotethical target system consisting of three components (yellow rectangles) which are logically interconnected (dashed lines) to realize the system's concerns. In order to introduce context aware behaviours in such a system, context sensing entities and context adaptation entities are defined. The former set consists of the definition of two contexts (green rectangles) which crosscut the system components in order to retrieve contextual information from them (green arrows) and two monitors that "observe" the contextual information retrieved by the contexts in order to recognize states of interest. When the system enters in a context state which is interesting for a monitor it automatically notifies such event to the adapter to which it is associated (blue arrow). For the sake of simplicity, in this example, only an adapter is defined (violet rectangle). Depending on the kind of trigger and on the contextual information retrieved, the monitor selects a proper adaptation layer containing the adaptation mechanisms to apply with respect to the target system. The applied layer will also crosscut the system components introducing mechanisms (i.e. bindings and inserts) enabling the system to properly react to the current context.

Several issues arise when *AdaptationLayer* has to be implemented concerning the fact that, depending on the adopted model, different layers consisting of adaptation mechanisms affecting the same entities (different bindings of the same entities, etc.), can possibly be activated at the same time. Many policies already exists aimed at properly handling the activation/deactivation of conflictual layers [80,81,82]. We therefore do not further specify how this issue should be addressed, leaving to the designer/developer the possibility to choose the solution he/she finds more suitable to his/her purposes.

As an example consider a PDA application able to share information with different kind of terminals exploiting the available network infrastructures that can go from wifi to bluetooth connections. A possible context aware enhancement can consist on the introduction of secure communication protocols that can be used by the application in order to communicate when the currently available network infrastructure is possibly insecure. Once this contextual information has been detected a proper adaptation mechanism should be applied that automatically changes the communication protocol. Such adaptation mechanism can be modeled by means of an adaptation layer consisting of a set of inserts, aimed at introducing structural end behavioural characteristics to the system components in order to implement the desired secure communication protocol,

and a binding aimed at substituting the references to the default communication protocol with a reference to the new one. Such adaptation layer can finally be triggered by the context state capturing the contextual information of being exploiting an insecure network and remains active until the related context state is deactivated.

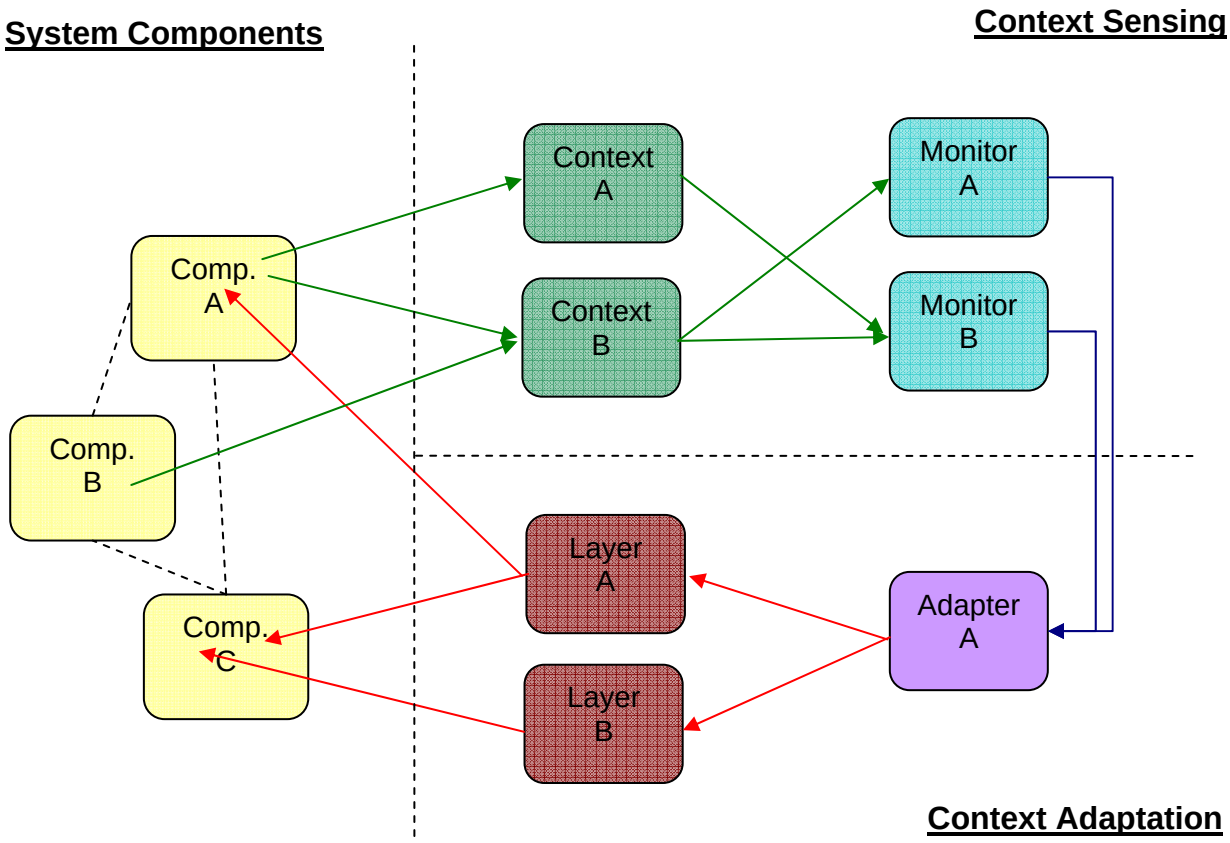

Figure 14: A graphical representation of a context awareness architecture

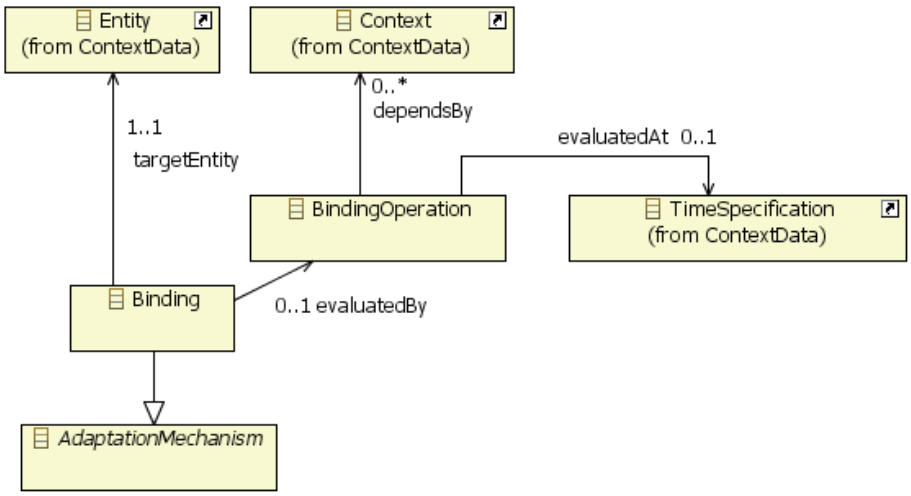

Figure 15: Context Aware Binding Model

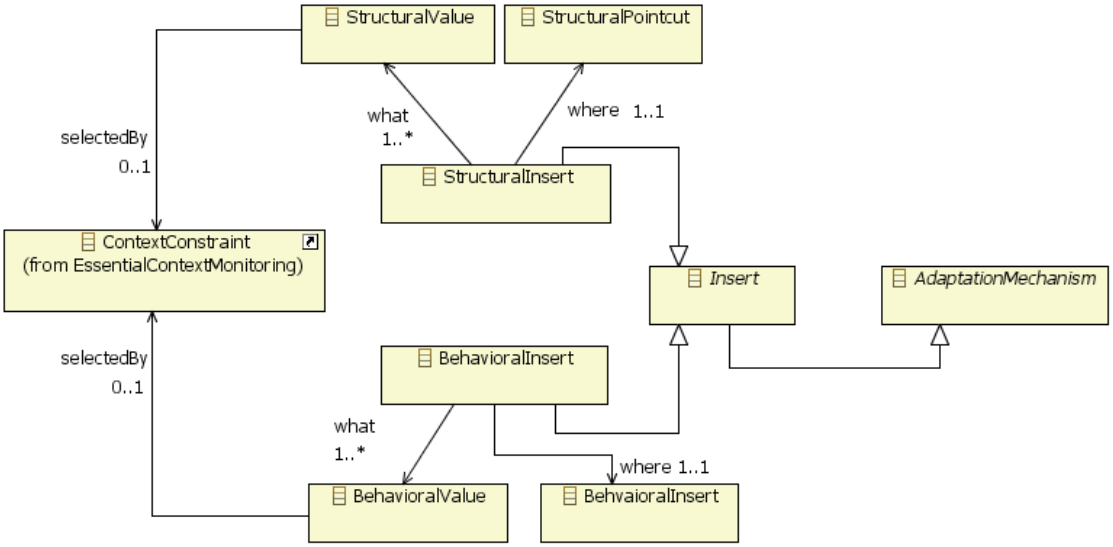

Figure 16: Insert Model

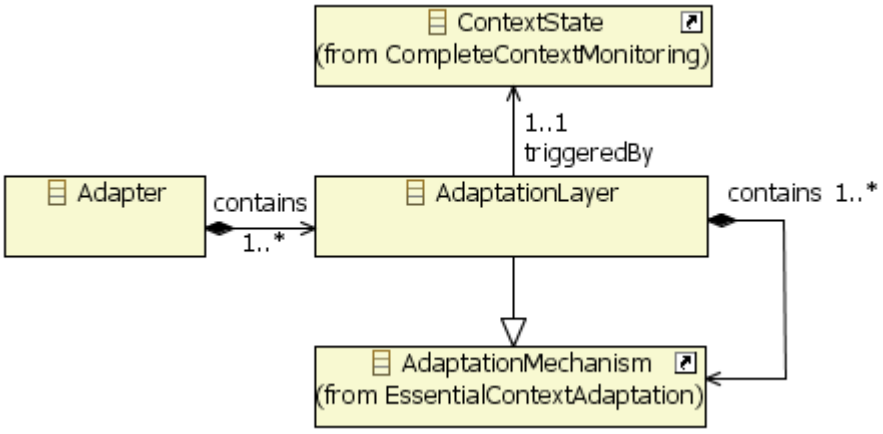

Figure 17: The Adapter Model

Concept name	**Semantics**	
AdaptationMechanism	Abstract concept which generically represents an adaptation mechanism exploitable to react to context changes. Adaptation mechanisms can be active or not with respect to a target system. When an adaptation mechanism is active it is supposed to change the standard behavior of the target system in order to react to the context change.	
	Attributes & Associations	**Semantics**
	Name	Each adaptation mechanism is uniquely identified by a name

	Active	Boolean attribute which is true when the *AdaptationMechanism* is active, false otherwise.
Concept name	**Semantics**	
Binding	Concept representing a pair consisting of an entity and a set of one or more values. When a binding is activated it instantaneously provides a value for the specified target entities. The value bound to a context entity may depend on its context so that it has to be evaluated by a binding operation.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the *AdaptationMechanism* concept
	Active	Inherited from the *AdaptationMechanism* concept
	targetEntity	Association which refers to the entity affected by the binding
	evaluatedBy	Association which refers to the binding operation that have to be exploited to choose the value to be bound
Concept name	**Semantics**	
BindingOperation	Concept which models an operation exploited by a binding to choose the value that has to be bound to a target entity. A binding operation may depends on one ore more contexts both past or presents through the *evaluatedAt* relation.	
	Attributes & Associations	**Semantics**
	dependsBy	Association which relates a binding operation with the Contexts it refers to
	evaluatedAt	Association which relates a binding operation with a time specification
Concept name	**Semantics**	
Insert	Abstract concept representing the introduction of new structural or behavioral elements into an already existing entity. It consists of a specification of the value that must be inserted, and of the point within some application entity where it must be inserted.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the *AdaptationMechanism* concept
	Active	Inherited from the *AdaptationMechanism* concept
Concept name	**Semantics**	
StructuralInsert	Concept representing an Insert aimed at introducing structural elements into an already existing entity. It corresponds to the concept of intertype declaration used in Aspect Oriented Programming. A structural insert is associated with a structural pointcut, which specifies the part of the static structure of the application which is affected by the mechanism, and set of structural values, which is used to specify the structural elements that must be introduced. This mechanism could be	

	used, for example, to introduce additional parameters within a service invocation or new components that provide additional functionalities.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the *AdaptationMechanism* concept
	Active	Inherited from the *AdaptationMechanism* concept
	Where	Association referring to the structural pointcut that indicates where the structural insert has to be performed
	What	Association referring to the structural value that indicates the value that the structural insert has to introduce in the point specified by the *where* association
Concept name	**Semantics**	
StructuralPointcut	Concept which specifies the part of the static structure of the application which is affected by a structural insert. It is exploited to indicate where a structural insert has to be performed.	
Concept name	**Semantics**	
StructuralValue	Concept which specifies the structural elements that must be introduced by a structural insert. More than one value can be associated with a given insert. In this case, to select the value to be inserted, each value in the specified set is associated with a context constraint through a *selectedBy* association	
	Attributes & Associations	**Semantics**
	selectedBy	Association which specifies the ContextConstraints that may drive the selection of a proper StructuralValue.
Concept name	**Semantics**	
BehavioralInsert	Concept representing an Insert aimed to introduce behavioral elements into an already existing entity. It corresponds to the concept of *advice* used in AOP. It is associated with a behavioral pointcut corresponding to the join point concept of AOP and is used to specify where, in the application dynamics, an additional behavior must be introduced, and with set of behavioural values which is used to specify the additional behavior itself.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the *AdaptationMechanism* concept
	Active	Inherited from the *AdaptationMechanism* concept
	Where	Association which refers to the

		behavioural pointcut that indicates where, in the target system dynamic, the behavioural insert has to be performed
	What	Association referring to the behavioural value that indicates the behavior that the behavioural insert has to introduce in the point specified by the *where* association
Concept name	**Semantics**	
BehavioralPointcut	corresponds to the join point concept of AOP and is used to specify where, in the target application dynamics, an additional behavior must be introduced	
	Attributes & Associations	**Semantics**
Concept name	**Semantics**	
BehavioralValue	Concept specifying the behavior that has to be introduced with respect to a behavioural insert. More than one behavior can be associated with a given insert. In this case, to select the behavior to be inserted, each behavior in the specified set is associated with a context constraint through a *selectedBy* association	
	Attributes & Associations	**Semantic**
	selectedBy	Association which specifies the context constraints that may drive the selection of a proper behavioural value.
Concept name	**Semantics**	
AdaptationLayer	This concept realizes the *AdaptationMechanism* abstract concept and represents a container for logically related adaptation mechanisms such as *Binding*, *Insert* and other *AdaptationLayers* through the *contains* aggregation with the *AdaptationMechanism* concept. An adaptation layer has to be considered active until one or more of the related context states are active. When an adaptation layer is activated all the related adaptation mechanisms are activated in turn. Otherwise, as soon as an adaptation layer is no longer active, its adaptation mechanisms are removed.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the *AdaptationMechanism* concept
	Active	Inherited from the *AdaptationMechanism* concept
	contains	An adaptation layer consists of a composition of other adaptation mechanism through the contains aggregation
	triggeredBy	Association relating an adaptation layer with one or more context states. Whenever one of the related

		context state is activated the adaptation layer is activated too and as a consequence all of its associated adaptation mechanisms.
Concept name	**Semantics**	
Adapter	Concept which acts as container for logically related adaptation layers. It represents an entity which receives signals by one or more logically related monitor whenever one of their context state goes active. When a monitor detects the activation of a context state it notifies the event to the interested adapters which cause the activation of the related adaptation layers that respectively cause the activation of their adaptation mechanisms.	
	Attributes & Associations	**Semantics**
	Name	Each *Monitor* is identified by a unique name
	contains	Aggregation which relates an Adapter with the *AdaptationLayers* it contains.

Figure 18: Context Adaptation concepts

3.4 A comparison with different approaches

In this section we would like to point out how our conceptual model relates to the other existing approaches for context awareness modeling and programming. To this end we have chosen some of the most representative works described in the 2nd chapter and tried to find out differences and similarities. We would first provide an informal demonstration of how our CMD tries to address the issues introduced in [10], namely:

- C*ontext Information Exhibits a Range of Temporal Characteristics*;
- *Context Information is Imperfect;*
- *Context Has Many Alternative Representations*;
- *Context Information is highly interrelated.*

In our conceptual model two mechanisms have been defined to model the range of temporal characteristics context information exhibits: *EventConstraint* and *TimeSpecification*. An event constraint captures precise pattern of events which are considered of interest for a given context so that, depending on the expressiveness of the event language used to define it, it can refer both to past or present events. The *TimeSpecification* concept instead can be used in the adaptation mechanisms (binding, insert, etc.) to refer to past context. It covers both the definition of precise time instant (clock time) and the definition of a temporal instant related to the occurrence of a certain context event.

In our conceptual model there is not an explicit representation of the grade of Imperfection of contextual information. That is because we think this is an element not generic enough to have a direct counterpart in a conceptual model. It can be modeled, as an example, with a context attribute representing a kind of contextual information a context entity should provide.

The gap between the information provided by sensor outputs and the level of information that is useful to applications can be addressed by the monitors. A Monitor is in fact an entity that araises the abstraction level of the context data abstraction from context information to context-state passing trhough context constraints. As explained in the 3.2 section, the *Monitor* concept represents that entity which continuously observes the contextual information provided by context entities (i.e. physical or logical sensors) detecting significant context-states, possibly inferring them, and providing more abstract contextual information to the related adapters that will finally use them in order to decide the adaptation layer to apply.

Finally, the modeling of the high interrelation that may occur between different context information is addressed by the composition relation that the *Context* concept has got with itself thought the use of the *CompositeContext* concept.

Once pointed out that our conceptual model addresses the requirements list proposed in [10] we would evidence how it is suitable to represent different context modeling approaches. To this end we take as reference the classification of context modeling approaches proposed in [5].

Key-value approaches can be easily modeled in our conceptual meta-model exploiting *StateBasedContext*, *ContextAttribute,* and *Source* concepts.

The same considerations hold for *markup-scheme* models where the composite relation a *StateBasedContext* has got with itself, throughout the *CompositeContext*, can be exploited to model the hierarchical nature of markup data structures.

The *graphical models* category is composed by heterogeneous approaches, hence it is not possible to provide a generic argumentation which demostrates our CDM can be exploitable to model them all. However we take as representative the modeling approaches introduced in [10] and [9] which relies on this category thus pointing out how those elements can be modeled with the concepts belonging to our conceptual framework. The modeling approach proposed in [10] consists of a graphical model where context information is structured around a set of entities which are conceptually similar to the context-entities of our model. Properties of entities are represented by

attributes. An entity is linked to its attributes and other entities by uni-directional relationships called associations. Therefore, in this model, a context description can be thought as a set of associations about an entity. Associations are then classified in *static vs. dynamic* rather than *simple vs. composite*. Static associations remain fixed over the lifetime of the entity that owns them while dynamic associations can be sub-categorized in: *sensed*, which are those whose values are directly obtained by the system sensors; *derived*, which are obtained from one or more other associations using a derivation function that may range in complexity from a single mathematical calculation to a complex AI algorithm; *profiled*, that are those information supplied by the user. An association is simple if each entity participating as owner of the association participates no more than once in this role. An example of this type of association is the named association of Person Figure 19 while the composite association can be refined in: *collection*, used to represent the fact that the owning entity can simultaneously be associated with several attribute values and/or entities (i.e. people may work with many other people); *alternatives*, which differ from collections in that they describe alternative possibilities that can be considered to be logically linked by the 'or' operator rather than the 'and' operator; *temporal*, where a temporal association is also associated with a set of alternative values, but each of these is attached to a given time interval. This type of information can be viewed as a function mapping each point in time to a unique value.

A mapping can be found between our conceptual model and the described approach, which consider associations as *StateBasedContext*s over the entities that own them. *Static* and *dynamic* associations can be modeled as a ContextAttribute: *sensed* associations directly take their value by the related sources which represent the involved sensors; the derivation function of *derived* associations can be modeled by a functionOf relation between ContextAttributes, *profiled* association which are those information supplied by the user are probably redundant because the user itself can be modeled as a context-entity rather than a source. *Collection* and *alternative* associations can be modeled exploiting the compositional relation that a Context, thus even a StateBasedContext, has with itself. Finally, temporal specification can be modeled exploiting the *TimeSpecification* concept thus identifying, when needed, the precise time instant where a certain event has occurred. In [9] the same authors propose an extension of their work in which they introduce the possibility to capture histories of events related to dynamic associations. Another extension is fact

dependencies, which represent a special type of relationship between facts, where a change in one fact leads automatically to a change in another fact. In our conceptual model histories of events are represented by the *EventConstraint* concept that, we remind, is defined as a pattern of events. This constraint holds at some time instant if the pattern can be identified within the history of events that have occurred up to that instant in the associated (through the uses association) *EventBasedContext*.

Fact dependencies can instead be considered special kind of association between Contexts as made for collection and alternative associations. Hence, our conceptual model encompasses the context modeling approach presented in [10,9] which can thus be considered one of its possible instantiations.

Object oriented models find a direct mapping with the *Context* concept. In these approaches the details of context processing is encapsulated at an object level and hence hidden from other components. The same function is addressed by the *Context* concept which acts as interface between the context entities complexity and the context monitoring entities thus exposing only those information that are of interest for the context-monitoring and context-adaptation concerns.

Because object-oriented modeling is generally accepted as a practical ontology specification: "*ontologies are meant to describe and explain the world while object model are meant to describe that part of the world whose representation has to be managed*" [83,84] the same consideration holds for ontology based models.

Once pointed out how our conceptual framework can be exploited to represent the entities involved in the existing context modeling approaches we would finally compare it with respect to other conceptual frameworks as those introduced in [85]. As an example, the Context tree depicted in Figure 1, which relates to an ontological way of defining contextual information, can be easily defined by means of a set of composite contexts representing the global *context* and *aspects* a set of state based context representing *facets* (i.e. performing time, history, location, business object, actor and organizational unit contextual information). Each of these state based contexts can finally be defined by a set of context attributes representing the facets' attributes (i.e. Duration, Day of the Week, etc.).

In [85] Dey et al. present a conceptual framework that separates the acquisition and representation of context from the delivery and reaction to context by a context-aware application. This approach is very similar to the one adopted in our conceptual model and it relies on the common agreement that by separating how context is acquired from

how it is used, applications can use contextual information without worrying about the details of a sensor and how to acquire context from it. However these details shouldn't be completely hidden and can be obtained if needed.

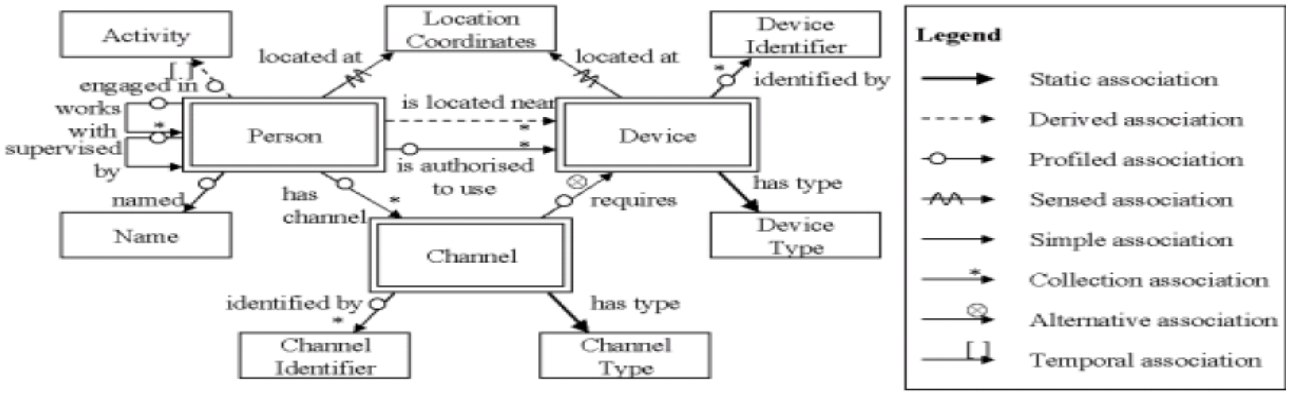

Figure 19: An example of context modeling taken from [10]

The conceptual framework presented in [85] consists in the definition of five entities: Context Widgets, Interpreters, Aggregators, Services and Discoverers.

- *Context Widgets*, provide a separation of concerns by hiding the complexity of actual sensors used from the application. They abstract context information to suit the expected need of applications thus providing reusable and customizable building blocks of context sensing. However, while context widgets support a uniform interface that allows applications to acquire context, they do not completely hide the underlying details of how context is being acquired from sensor. In fact these may include details on the type of sensor used, how data is acquired from the sensor, and the resolution and accuracy of the sensor. The default behavior shouldn't be to provide these details to applications. However, applications can request this information if desired;
- *Interpreters* refer to the process of raising the level of abstraction of a piece of context. For example simple inference or derivation may be used to transform geographical coordinates into street names maybe using a geographic information database. Complex inference using multiple pieces of context also provides higher-level information. An interpreter typically takes information from one or more context sources and produces a new piece of context-information. Interpretation of context has usually been performed by applications. By separating the interpretation out from applications, reuse of interpreters by multiple applications and widgets is supported;

- *Aggregators* refers to collecting multiple pieces of context information that are logically related into a common repository. Aggregators gather logically related information relevant for applications and make it available within a single software component. By collecting logically related information, aggregators provide an abstraction that helps handle consistently separate pieces of context information, and offer a "one-stop shop" for applications;
- *Services* are components in the framework that execute actions on behalf of applications. A context service is an analog to the context widget. Whereas the context widget is responsible for retrieving state information about the environment from a sensor, the context service is responsible for controlling or changing state information in the environment using an actuator. As with widgets, applications do not need to understand the details of how the services is being performed in order to use them;
- *Discoverers* are the final component in the conceptual framework presented in [85]. They are responsible for maintaining a registry of what capabilities exist in the framework. This includes knowing what widgets, interpreters, aggregators and services are currently available for use by applications. When any of these components are started, it notifies a discoverer of its presence and capabilities and how to contact that component. Widgets indicate what kind of context they can provide. Interpreters indicate what interpretations they can perform. Aggregators indicate what entity they represent and the type of context they can provide about that entity. Services indicate what context-aware service they can provide and the type of context and information required to execute that service. Applications and other context component use the discoverers to locate context components that are of interest to them.

The conceptual model presented by Dey et al. share with our model the strong distinction between context monitoring and context adaptation concerns. The context widget concept can be easily mapped to our *Context* concept which represents the context information, both state or event based, brought by a context-entity. Even our *Context* concept also encapsulates and hides the actual logical or physical source of the context information.

Interpreters and *Aggregators* can instead be mapped to the *Monitor* concept which addresses both functions of raising the context information's level of abstraction and gathering logically related context information thus making it available to the

Adapters. *Services* can be considered equivalent with the *Adapter* concept which is the entity aimed at introducing *AdaptationMechanisms* to react to the context changes detected by related *Monitors*. Our conceptual model does not provide a concept comparable with the *Discoverer* concept. That is because we consider the function of maintaining a registry of what capabilities exist in the framework at a given instant (what contexts, monitors and adapters are currently available) strictly dependent on the kind of target application in which the conceptual model is instantiated and used.

Both the mentioned approaches to dynamically adapt the behaviour of a system namely aspect orientation (section 2.4.1) and dynamic binding (section 2.4.2) are represented in our conceptual model. The former is represented by the structural end behavioral insert concepts which respectively make possible to model the context driven introduction of new behavioral and structural features into target system's entities. The latter is instead represented by the binding concept which makes possible to model the context driven rearrangement of the target system's components. These two mechanisms are finally managed by means of layers grouping logically related behavioral variations, which is the same approach dopted in [57] by Costanza et al.

Finally we would like to point out the differences and similarities with the ContextUML metamodel presented in [7] which is somehow similar to our approach. Smilarly to the metamodel proposed by Sheng et al. in our conceptual model we have taken well separated the two aspects of *context modeling* and *context awareness modeling*. However with respect to the ContextUML, in our conceptual model the context awareness aspect is itself defined by menas of two well separated concerns, namely: *context adaptation triggering* and *context driven adaptation* so that the separation of concerns is improved. ContextUML only takes into account what we have called state based contexts while through our conceptual model it is possible to define a context both by means of statical and behavioral features such as the events internal or external to a target system. As a consequence, while ContextUML makes possible to define context constraint by means of logical predicates over the structural features composing a context, through our conceptual model context constraints can also take into account the occurrence of complex pattern of events. Temporal specifications make possible to refer to the contextual information brought by past contexts while in [7] this possibility is not mentioned. In ContextUML context

driven adaptation is performed by means of dynamic binding of web service interfaces with different implementation. Through our conceptual model adaptation can be modeled by means of dynamic binding and aspect oriented approaches which also make possible to introduce new structural and behavioral features to the components of a target systems. Finally, while ContextUML has been specifically designed for the modeling of context aware web services our conceptual model does not directly refer to a specific technology. So that it can be taken as reference both for the modeling of context aware web services or for the modeling of context aware traditional components.

4 The Model Driven Development Paradigm

The Model-driven engineering (MDE) [86,87,88] is a discipline in computer science that relies on models as first class entities and that aims to develop, maintain and evolve software by performing model transformations. MDE, also referred to as Model Driven Development (MDD), is embraced by various organization and companies, including OMG [18], IBM [89] and Microsoft [90] and encompasses a wide variety of different techniques, including OMG's Model Driven Architecture (MDA). These paradigms are aimed at defining an approach to IT systems specification that separates the specification of system functionalities from the specification of their implementation with respect to a certain technological platform. They can thus be regarded as a natural continuation of the trend aimed at raising the level of abstraction at which the activity of computer programming takes place. In this way, developers, instead of use a programming language spelling out how a system is implemented, can directly use models to specify what system functionality is required and what architecture has to be used. A wide variety of tools exist, such as the Eclipse Generative Model Transformer (GMT)[91], the Eclipse Modeling Framework (EMF)[20], the Generic Modeling Environment (GME)[93], etc. [94], that enables the adaption of a model driven paradigm. In [87] Atkinson et al. summarize the key elements a MDD supporting infrastructure must define. These are:

1. the concepts that are available for the creation of models and the rules governing their use;
2. the notation to be used in the depiction of models;
3. how the elements of a model relate to real world elements;
4. concepts to facilitate dynamic user extensions to (1) and (2), and models created from them
5. concepts to facilitate the interchange of (1) and (2), and models created from them;
6. concepts to facilitate user defined mappings from models to other artifacts which may include: executable code, test suites or even performances analysis.

They also point out how an MDD Infrastructure shall be based on three main key foundations that have long record of success in software engineering.

1. visual modeling: is one of the technological foundations for a technological MDD support because it addresses the requirements (1)-(3) and makes effective use of human visual perception;

2. object orientation: because it supports flexible choice of modeling concepts (1) and extendable languages (4). Object-oriented languages enable language users to extend the set of available types which makes it generally accepted as one of the key foundations of model driven development;
3. the use of metalevel descriptions: it addresses the issues defined by requirements (5)-(6). Beyond that, metalevel descriptions are also vital for supporting both static and dynamic aspects of requirement (4).

The OMG's MDA defines an architecture for models that addresses all these points thus providing a set of guidelines for structuring specification expressed as models.

In this thesis we consider as reference modeling framework the Model Driven Architecture and its most important constituent: the Meta Object Facility language (MOF) and the Unified Model Language (UML) that will be introduced in the next sections. Starting from this reference architecture we have designed and developed a MDA based framework for context awareness modeling and development which exploits a domain specific modeling language called CAMEL (Context Aware ModEling Language) that can be considered a UML heavy-weight extension (section 4.5) implementing the conceptual domain model described in the chapter above.

4.1 MDA Concepts

The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the system's operations from the details about the way the system should exploits the capabilities of its platform.

MDA provides an approach for, and enables tools to be provided for:

- Specifying a system independently of the platform that may supports it;
- Specifying platforms;
- Choosing a particular platform for the system, and
- Transforming the system specification into one for a particular platform.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns. At the core of OMG's Model Driven Architecture are the concepts of *models*; *metamodels* defining the abstract languages in which the models are captured; and *transformations* that take a set of models and produce another set of models from them [95]. Figure 20 depicts the relationships between the MDA major concepts.

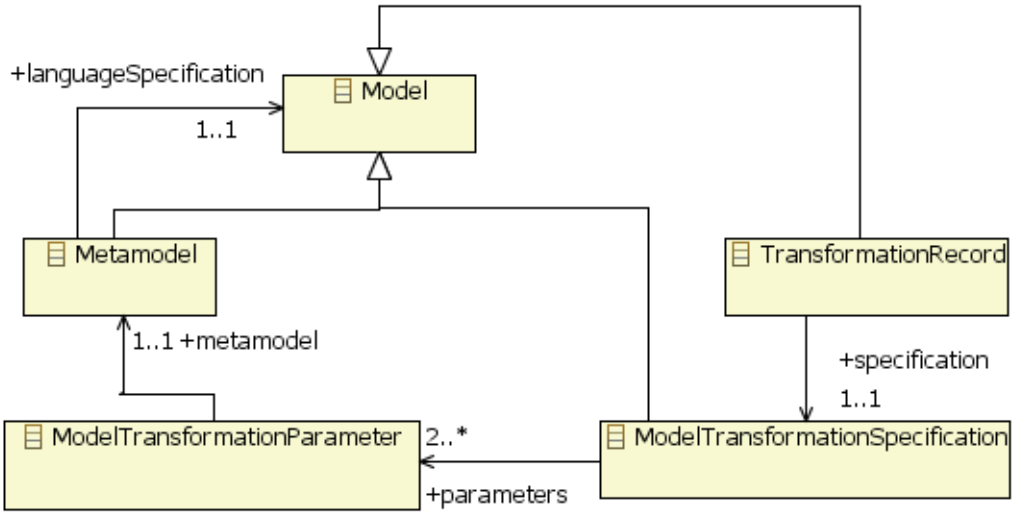

Figure 20: Overview of the reference model core

4.1.1 Model

A model is an abstraction representing something. It does so with a specific purpose in mind. The model is related to the thing by an explicit or implicit isomorphism. Models in the context of the MDA Foundation Model are instances of MOF meta-models and therefore consist of model elements and links between them. Model elements in turn can hold attribute values which can be of primitive types, such as Integer, Boolean or String. The relationship between a model and its abstract language specification is captured in the model shown in Figure 20 by the association with the role named *languageSpecification*.

Figure 21 depicts the composite nature of a model which is common to most well-known modeling and meta-modeling paradigms, not only the OMG's MOF.

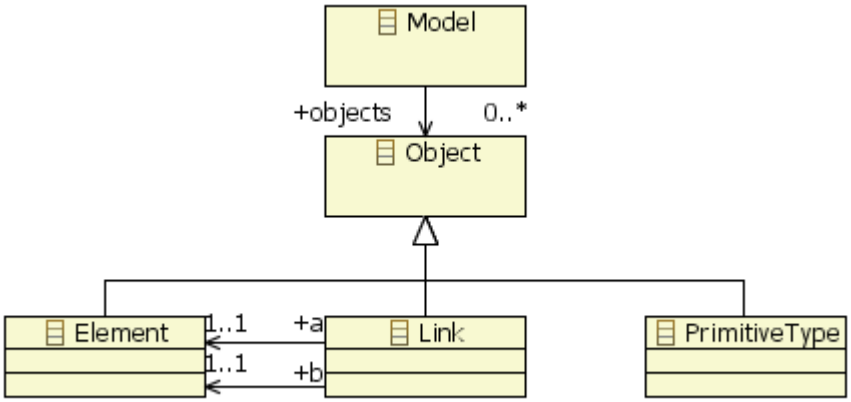

Figure 21: The model costituents

The main constituents of models are Elements that can be associated with one another using proper links. Elements and links are both typed. In MOF the type of an Element is itself an element whose type is a MOF:Association.

4.1.2 Metamodel

A metamodel is a special kind of model that specifies the abstract syntax of a modeling language. It can be understood as the representation of the class of all models expressed in that language. Metamodel in the context of MDA are expressed using MOF which is described in section 4.3.

4.1.3 Model Transformation

In [96] Mens et al. define a model transformation as "*the automatic generation of a target model from a source model, according to a transformation definition*". A *transformation definition* is defined as "*a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language*". A transformation rule is finally defined as "*the description of how one or more constructs in the source language can be transformed into one or more constructs in the target language*".

Mens et al. also introduce a taxonomy of the possible model transformations. The first distinction among model transformations can be done between *endogenous* and *exogenous* transformations. The former category consists of those transformations in which both source and target models are defined by the same meta-modeling language while the latter consists of those transformations in which the two models are defined by different meta-modeling languages. Typical examples of exogenous transformations are the *reverse engineering* of a lower-level specification (code) to an higher-level one (UML), or the *migration* from a program written in one language to another, but keeping the same level of abstraction. The *refactoring* of a software, usually done to improve certain software quality characteristics, is instead a typical example of endogenous transformations.

The second distinction is made between *horizontal* and *vertical* transformations. A horizontal transformation is a transformation where the source and target models reside at the same abstraction level while a vertical transformation is a transformation where the source and target models reside at different abstraction levels. An example of horizontal transformation is the refactoring, which is a process aimed at improving the some characteristics of a source model (i.e. maintainability, etc.) without changing its

overall semantic. A typical example of vertical transformation is the refinement where a specification is gradually refined into a full fledged implementation by the means of successive refinement steps that add more concrete details [97,98]. The final distinction is made between *Syntactical* and *Semantical* transformations which distinguish the model transformations that merely transform the syntax by the more sophisticated transformations that also take the semantics of the model into account.

In the OMG's MDA [95] a model transformation specification determines how a set of output models results from a set of input models. Figure 22 depicts at a very abstract level the internal structure of a model transformation specification. Regardless of the particular implementation technology used for a model transformation specification, it can be safely assumed that is made up of individual elements that can be distinguished. The meta-models in the roles of language of the input models and language of the output models determine the languages in which the input and output models are expressed. These languages sets are ordered in a list of formal parameters *ModelTransformationFormalParameters*. In other words, these associations define the input and output side of the signature of the model transformation.

This design assumes that a model transformation specification is mainly unidirectional. In the case when a transformation can be specified as working in both directions, it can be represented in this model as two unidirectional transformation specifications that just happen to have the input and output sides of their signature interchanged.

Each *ModelTransformationSpecification* element determines how a group of output objects results from a group of input objects. A model transformation is executed based on the model transformation specification and the model transformation elements it contains. The transformation binds a set of input models to the formal input parameters of the model transformation specification, and a set of output models to the formal output parameters of the model transformation specification.

In the workflow of a possible MDA based approach there may be several model transformations. The most common typically concerns the transformation of a Platform Independent Model (PIM) into a Platform Specific Model (PSM) given the model of a target platform. A model is platform independent when it is independent of the features of a platform of any particular type. Therefore a PIM describes the system and its functionalities but does not show details about its platform. The model of a platform can be provided that enables implementation of the system with the desired architectural qualities.

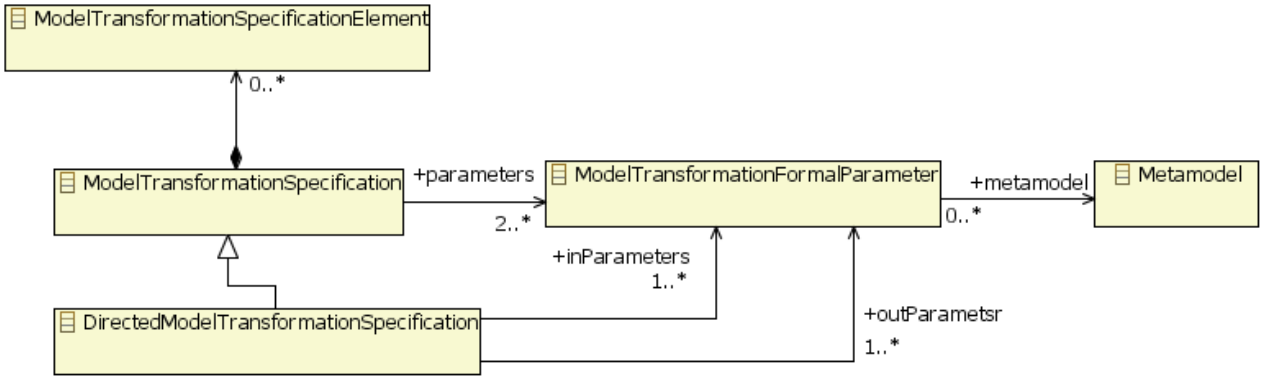

Figure 22: Internal Structure of a Transformation Specification

A platform specific model, produced by a proper transformation, is a model of the same system specified by the PIM but it also specifies how that system makes use of the chosen platform. A PSM may provide more or less detail, depending on its purpose. It will be an implementation if it provides all the information needed to construct a system or it may act as a PIM itself that is used for further refinement to a PSM that can be directly implemented. There may be several approaches aimed at transforming a model into another (i.e., model marking, meta-model transformation, etch.). For example Figure 23 depicts the workflow of a PIM to PSM model transformation passing through the use of their meta-models. A model is prepared using a platform independent language specified by a meta-model. A particular platform is chosen. A specification of a transformation for this platform is available or is prepared. This transformation specification is in terms of a mapping between meta-models. The mapping guides the transformation of the PIM to produce the PSM.

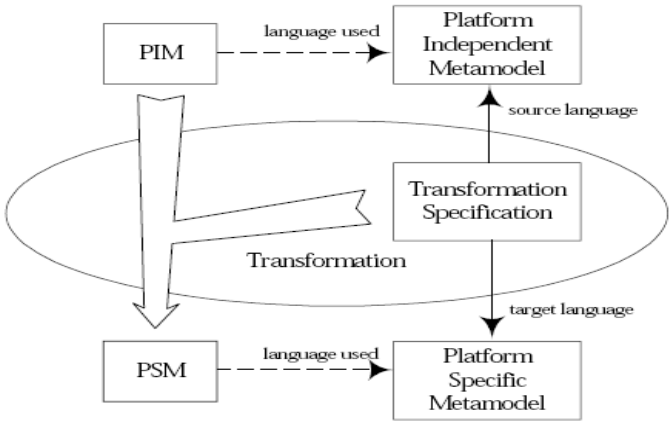

Figure 23: workflow of a meta-model based model transformation

4.2 OMG Modeling Infrastructure

The central theme of the MOF approach to metadata management is extensibility. The aim is to provide a framework supporting any kind of metadata, and that allows new kinds to be added as required. In order to achieve this, the MOF has a layered metadata architecture that is based on the classical four layer metamodeling architecture popular within standards communities such as ISO and CDIF.

Figure 24 illustrates the traditional four layer infrastructure that underpins the MDA technologies; namely the UML [74] and the MOF [78]. This infrastructure consists of a hierarchy of model levels, each, except the top, characterized as an instance of the level above. The bottom level also referred to as M0 is said to hold the "user Data", i.e., the actual data objects the software is designed to manipulate.

The next level, M1 is said to hold a "model" of the M0 user data. This is the level at which user models reside. Level M2 is said to hold a "model" of the information at M1. Since it is a model of a user model, it is often referred to as a meta-model. Finally, level M3 is said to hold a model of the information at M2, and hence is often characterized as the meta-meta-model. For historical reason it is also referred as the MOF (Meta Object Facility). In a top-down view of this hierarchy we can think at the M3 level, the MOF, as the language used to define meta-models, such as UML. In the M2 level, a meta-model as the UML can be seen as the language used to define user models (M1) which consist of language to represent user data (M0).

This layered architecture has the advantage of easily accommodating new modeling standards as instances of MOF at the M2 level. MOF aware tools can thus support the manipulation of new modeling standards and enable information interchange across MOF compatible modeling standards.

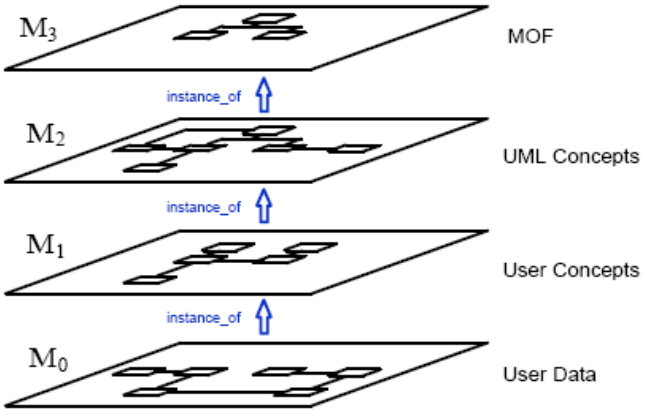

Figure 24: Traditional OMG Modeling Infrastructure

4.3 MOF: The OMG's Meta Object Facility

4.3.1 MOF Meta-modeling constructs

This section introduces the MOF's core metamodeling constructs for defining metamodels. MOF metamodeling is primarily about defining information models for metadata.

The MOF uses an object modeling framework that is essentially defined as a subset of the UML core correspondent to the Core package of the UML infrastructure (section 4.4). In a nutshell, the four main concepts belonging to the Core package and exploited by the MOF model are:

- Classes, which model MOF metaobjects;
- Associations, which model binary relationships between metaobjects;
- DataTypes, which model other data (e.g., primitive types, external types, etc.)
- Packages, which modularize the models.

4.3.1.1 Classes

Classes are type description of "first class instance" MOF metaobjects. Classes defined at the M2 level logically have instances at the M1 level. These instances have object indentity, state and behavior. The state and behavior of the M1 level instances are defined by the M2 level Class in the context of the common information and computational models defined by the MOF specification.

MOF Classes can have three kinds of features: Attributes, Operations and References. They can also contain Exceptions, Constants, DataTypes, Constraints and other elements.

Attributes

An attribute defines a notational slot or value holder, typically in each instance of its Class. An attribute is characterized by the properties depicted in Figure 25:

Property	Description
Name	Unique in the scope of the Attribute's Class.
Type	May be a Class or a DataType.
isChangeable	Flag which determines whether the client is provided with an explicit operation to set the attribute's value.

isDerived	Flag which determines whether the contents of the notational value holder is part of the explicit state of a Class instance or is derived from other state.
Multiplicity	An Attribute or Parameter may be optional-valued, single-valued, or multi-valued depending on its multiplicity specification. This consists of three parts: • The "lower" and "upper" fields place bounds on the number of elements in the Attribute or Parameter value. The lower bound may be zero and the upper may be "unbounded"; • The "is_ordered" flag says whether the order of values in a holder has semantic significance. For example, if an Attribute is ordered, the order of the individual values in an instance of the Attribute will be preserved. • The "is_unique" flag says whether instances with equal value are allowed in the given Attribute or Parameter. The meaning of "equal value" depends on the base type of the Attribute or Parameter. See Section 4.4, "Semantics of Equality for MOF Values," on page 4-3, and Section 5.3.3, "Value Types and Equality in the IDL Mapping," on page 5-11 for additional information.

Figure 25: The MOF Attributes' properties

Operations

Operations are "hooks" for accessing behavior associated with a Class. Operations do not actually specify the behavior or the methods that implement the behavior. Instead they simply specify the names and type signatures by which the behavior is invoked.

Operations is characterized by properties depicted in Figure 26:

Property	Description
Name	Unique in the scope of the Attribute's Class.
List of positional parameters having the following properties	
ParameterName	
ParameterType	May be denoted by a Class or a DataType
ParameterDirection	determines whether actual arguments are passed from client to server, server to client, or both.
Multiplicity	See Attribute Multiplicity

An optional return type
A list of Exceptions that can be raised by an invocation

Figure 26: The MOF Operations' properties

Classes Generalization

The MOF allows Classes to inherit from one or more other Classes. Following the lead of UML, the MOF Model uses the verb "to generalize" to describe the inheritance relationship (i.e., a super-Class generalizes a sub-Class).

The meaning of MOF Class generalization is similar to generalization in UML and to interface inheritance in CORBA IDL. The sub-Class inherits all of the contents of its super-Classes (i.e., all of the super-Classes Attributes, Operations and References, and all nested DataTypes, Exceptions and Constants). Any explicit Constraints that apply to a super-Class and any implicit behavior for the super-Class apply equally to the sub-Class. At the M1 level, an instance of an M2-level Class is type substitutable for instances of its M2-level super-Classes.

4.3.2 **Association**

Associations are the MOF Model's primary construct for expressing the relationships in a metamodel. At the M1 level, an M2 level MOF Association defines relationships (links) between pair of instances of Classes. Conceptually, these links do not have object identity, and therefore cannot have Attribute or Operations.

Each MOF Association contains precisely two Association Ends describing the two ends of links. The Association Ends is characterized by the properties depicted in Figure 27:

Property	Description
A name for the End	This is unique within the Association
A type for the End	This must be a Class
Multiplicity Specification	Each Association End has a multiplicity specification. While these are conceptually similar to Attribute and Operation multiplicities, there are some important differences: • An Association End multiplicity does not apply to the entire link set. Instead, it applies to projections of the link set for the possible values of the "other" end of a link; • Since duplicate links are disallowed link sets, "is_unique" is implicitly TRUE. The check for duplicate links is based on equality of the

	instances that theyconnect; • The “lower” and “upper” bounds of an Association End constrain the number of instances in a projection; • The “is_ordered” flag for the Association End determines whether the projections from the other End have an ordering;
An Aggregation Specification	See the aggregation semantic on section 4.3.3
A navigability Setting	Controls whether References can be defined for the end
A “changeability” setting	Determines whether this end of a link can be updated “in place.”

Figure 27: The MOF Associations’ properties

4.3.3 Aggregation Semantic

In MOF metamodel Classes and DataTypes can be related to other Classes using Associations or Attributes. In both cases aspects of the behavior of the relationships can be described as aggregation semantics. The MOF supports two kinds of aggregation for relationships between instances (i.e., “*composite*” and “*non-aggregate*”).

A non-aggregate relationship is a conceptually loose binding between instances with the following properties:

- there are no special restrictions on the multiplicity of the relationships;
- there are no special restrictions on the origin of the instances in the relationships;
- The relationships do not impact on the lifecycle semantics of related instances. In particular, deletion of an instance does not cause the deletion of related instances;

By the contrast a composite relationship is a conceptually stronger binding between instances with the following properties:

- A composite relationship is asymmetrical, with one end denoting “composite” or “whole” in the relationship and the other one denoting the “components” or “parts”;
- An instance cannot be a component of more than one composite at a time, under any composite relationship;

- An instance cannot be a component of itself, its components, its components' components and so on under any composite relationship;
- When a composite instance is deleted all of its components under any composite relationship are also deleted, and all of the components' components are deleted and so on.

The aggregation semantics of an Association is specified explicitly using the "aggregation" attribute of the AssociationEnds. In the case of a "composite" Association, the "aggregation" attribute of the "composite" AssociationEnd is set to true and the "aggregation" Attribute of the "component" AssociationEnd is set to false. Also, the multiplicity for the "composite" AssociationEnd is set to false.

The effective aggregation semantics for an Attribute depend on the type of the Attribute. For example:

- An Attribute whose type is expressed as a DataType has "non-aggregate" semantics;
- An Attribute whose type is expressed as a Class has "composite" semantics;

It is possible to use a DataType to encode the type of a Class. Doing this allows the metamodel to define an Attribute whose value or values are instances of a Class without incurring the overhead of "composite" semantics.

4.3.4 References

The MOF Model provides two constructs for modeling relationships between classes: Associations and Attributes. While MOF Associations and Attributes are similar from the information modeling standpoint, they have important differences from the standpoints of their computational models and their corresponding mapped interfaces.

Associations offer a "query-oriented" computational model. The user performs operations on an object that notionally encapsulates a collection of links:

- Advantage: The association objects allow the user to perform "global" queries over all relationships, not just those for a given object.
- Disadvantage: The client operations for accessing and updating relationships tend to be more complex.

Attributes offer a "navigation-oriented" computational model. The user typically performs get and set operations on an attribute.

- Advantage: The get and set style of interfaces are simpler, and tend to be more natural for typical metadata oriented applications that “traverse” a metadata graph.
- Disadvantage: Performing a “global” query over a relationship expressed as an Attribute is computationally intensive;

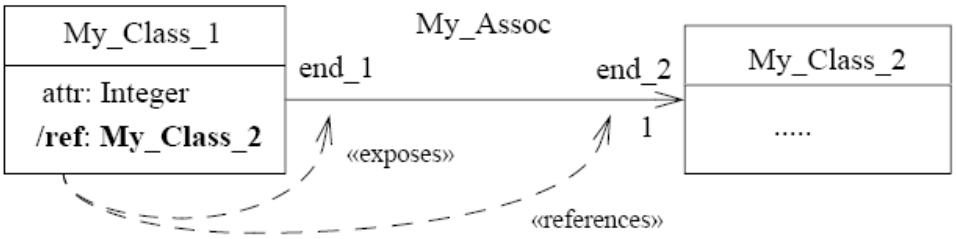

Figure 28: an example of a Reference

The MOF model provides an additional kind of Class feature called Reference that provides an alternative “Attribute like” view of Associations.

A reference is specified by giving the following:

- A name for the Reference in its Class;
- An “exposed” Association End in some Association whose type is this Class or a super-Class of this Class and
- A “referenced” Association End, which is the other end of the same Association

Defining a Reference in a Class causes the resulting interface to contain operations with signatures that are identical to those for an “equivalent” Attribute. However rather than operating on the values in an attribute slot of a Class instance, these operations access and update the Association, ore more precisely a projection of the Association.

Figure 28 depicts a Class called My_Class_1 that is related to My_Class_2 by the Association My_Assoc. My_Class_1 has an Attribute called “attr” whose type is Integer. In addition, it has a Reference called “ref” that references “end2” of the Association. This provides an API for “ref” that allows a user to access and update a My_Class_1 instance’s link to a My_Class_2 instance using get and set operations.

The example above shows a Reference that “exposes” an Association End with a multiplicity of “[1..1]”. References can actually expose ends with any valid multiplicity specification. The resulting Reference operations are similar to those for an Attribute with the same multiplicity. However, since MOF Associations do not allow duplicates, Association Ends and therefore References must always have their multiplicity “is_unique” flag set to true. There are some important restrictions on References:

- When the “is_navigable” property of an Association End is false, it is not legal to define a Reference that “references” that Association End.
- An M1 instance of a Class that “references” an Association cannot be used to make a link in an instance of the Association in a different extent. This restriction is described in Section 4.11.1, “The Reference Closure Rule,” on page 4-19.

4.3.5 The MOF Model

The Model of the MOF language is defined by means of the modeling elements defined in the section above. The MOF model itself defines a set of modelling elements, including the rules fore their use, by meta-models can be defined or extended. This focus on meta-models enables the MOF to provide a more domain-specific modelling environment for defining meta-models instead of a general purpose modelling environment. A well designed modelling tool or facility should be based on a meta-model that represents the modelling elements and the rules provided by the tool or facility. In this section we provide a brief description of the essential MOF model elements referring to the OMG’s MOF specification for further details [78]. The MOF model, as it is currently defined, consists of a single non-nested Package called “Model.” This Package explicitly imports the “PrimitiveTypes” Package so that it can use “Boolean,” “Integer,” and “String” PrimitiveType instances. The diagram Figure 29 depicts the Classes and Associations of the “Model” Package. To aid readability, Class features, Association End names, DataTypes, and other details have been omitted from the diagram.

The key abstract Classes in the MOF Model are as follows:

- *ModelElement* - this is the common base Class of all M3-level Classes in the MOF Model. Every ModelElement has a “name.”
- *Namespace* - this is the base Class for all M3-level Classes that need to act as containers in the MOF Model.
- *GeneralizableElement* - this is the base Class for all M3-level Classes that support “generalization” (i.e., inheritance).
- *TypedElement* - this is the base Class for M3-level Classes such as Attribute, Parameter, and Constant whose definition requires a type specification.
- *Classifier* - this is the base Class for all M3-level Classes that (notionally) define types. Examples of Classifier include Class and DataType.

The key Associations in the MOF Model are as follows:

- *Contains*: this Association relates a *ModelElement* to the *Namespace* that contains;
- *Generalizes*: this Association relates a *GeneralizableElement* to its ancestors (i.e., supertypes) and children (i.e., subtypes) in a model element inheritance graph. Note that a *GeneralizableElement* may not know about all of its subtypes.
- *IsOfType*: this Association relates a *TypedElement* to the Classifier that defines its type.
- *DependsOn* - this derived Association relates a *ModelElement* to others that its definition depends on. (It is derived from Contains, Generalizes, *IsOfType*, and other Associations not shown here.)

The most important relationship in the MOF Model is the *Contains* Association. Containment is a utility Association that is used to relate (for example) Classes to their Operations and Attributes, Operations to their Parameters, and so on. While the class diagram shows that only ModelElement objects which are subtypes of Namespace can contain any other ModelElements, the MOF Model restricts the legal containments to eliminate various nonsensical and problematical cases. A detailed specification of the MOF model's concepts can be found in [78], while in section 4.6.1 we provide a brief explanation of the MOF concepts which have been implemented and are then available in the Eclipse Modeling Framework [20].

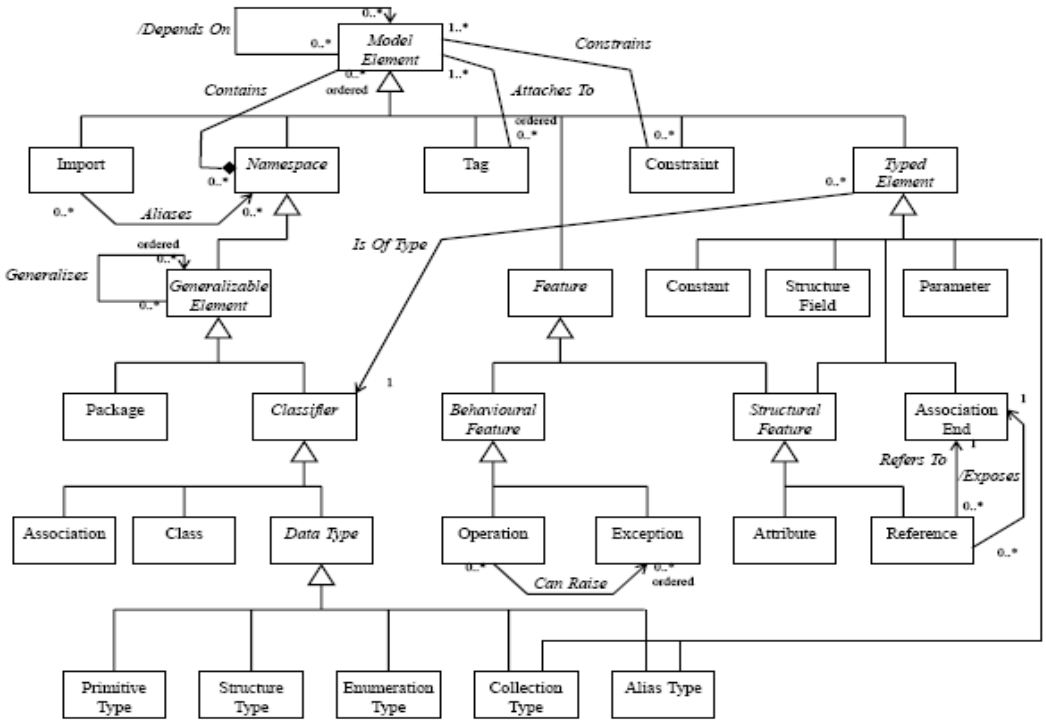

Figure 29: The MOF Model package

4.4 UML: The Unified Modelling Language

The Unified Modeling Language is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modelling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET). The OMG adopted the UML 1.1 specification in November 1997. Since then UML Revision Task Forces have produced several minor revisions, which fixed shortcomings and bugs with the first version of UML. In 2005 the OMG adopted the UML 2.0 major revision. There are four parts to the UML 2.x specification: The *Superstructure*, that defines the notation and semantics for diagrams and their model elements; the *Infrastructure*, that defines the core meta-model on which the *Superstructure* is based; the Object Constraint Language (OCL) for defining rules for model elements; and the UML Diagram Interchange that defines how UML 2 diagram layouts are exchanged. The current versions of these standards follow the UML Superstructure version 2.1.2, UML Infrastructure version 2.1.2, OCL version 2.0 and UML Diagram Interchange version 1.0.

Following the OMG's Modeling Infrastructure, the UML specification is defined using a meta-modeling approach (i.e., a metamodel is used to specify the model that comprise UML) that adapts formal specification techniques. While this approach lacks some of the rigor of a formal specification method it offers the advantages of being more intuitive and pragmatic for most implementers and practitioners.

The Infrastructure of the UML is defined by the *InfrastructureLibrary* package, which contains modelling elements aimed at satisfying the following design requirements:

- To define a meta-language core that can be reused to define a variety of meta-models, including UML and MOF;
- To architecturally align UML, MOF, and XMI so that model interchange is fully supported;
- To allow customization of UML through Profiles and creation of new languages (family of languages) based on the same metalanguage core as UML.

As shown in Figure 30, the *InfrastructureLibrary*, consists of two packages: *Core* and *Profiles*. The latter package defines the mechanisms that are used to customize meta-models while the former contains core concepts used when meta-modeling.

In order to facilitate reuse, the *Core* package is subdivided into a number of packages: *PrimitiveTypes*, *Abstractions*, *Basic*, and *Constructs* as depicted in Figure 31. Some of these packages are then further divided into even more fine-grained packages to make it possible to pick and choose the relevant parts when defining a new meta-model. Note, however, that choosing a specific package also implies choosing the dependent packages.

The package *PrimitiveTypes* simply contains a few predefined types that are commonly used when meta-modeling, and is designed specifically with the needs of UML and MOF in mind. Other meta-models may need other or overlapping sets of primitive types. There are minor differences in the design rationale for the other two packages. The package *Abstractions* mostly contains abstract meta-classes that are intended to be further specialized or that are expected to be commonly reused by many meta-models. Very few assumptions are made about the meta-models that may want to reuse this package. The package *Constructs*, on the other hand, mostly contains concrete meta-classes that lend themselves primarily to object-oriented modeling; this package in particular is reused by both MOF and UML, and represents a significant part of the work that has gone into aligning the two meta-models. The package *Basic* represents a few constructs that are used as the basis for the produced XMI for UML, MOF, and other meta-models based on the *InfrastructureLibrary*.

The *Core* package is therefore used to define the modelling constructs used to create metamodels. This is done through instantiation of meta-classes in the *InfrastructureLibrary* . While instantiation of meta-classes is carried out through MOF, the InfrastructureLibrary defines the actual meta-classes that are used to instantiate the elements of UML, MOF, CWM, and indeed the elements of the *InfrastructureLibrary* itself. In this respect, the *InfrastructureLibrary* is said to be self-describing, or reflective. The *Profiles* package depends on the *Core* package, and defines the mechanisms used to tailor existing meta-models, such as the UML, towards specific platforms, such as C++, CORBA, or EJB; or domains such as real-time, business objects, or software process modelling. One of the major goals of the *Infrastructure* has been to architecturally align UML and MOF. The first approach to accomplish this has been to define the common core, which is realized by the *Core* package, in such a way that the model elements are shared between UML and MOF.

The second approach has been to make sure that UML is defined as a model that is based on MOF used as a meta-model, as is illustrated in Figure 32. Note that MOF is used as the

meta-model for not only UML, but also for other languages such as CWM. An important aspect that deserves mentioning here is that every model element of UML is an instance of exactly one model element in MOF. Note that the *InfrastructureLibrary* is used at both the M2 and M3 meta-levels, since it is being reused by UML and MOF, respectively, as was shown in.

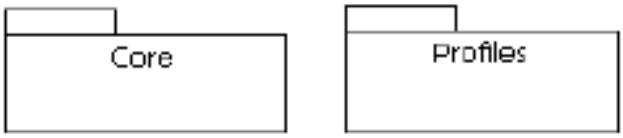

Figure 30: The InfrastructureLibrary packages

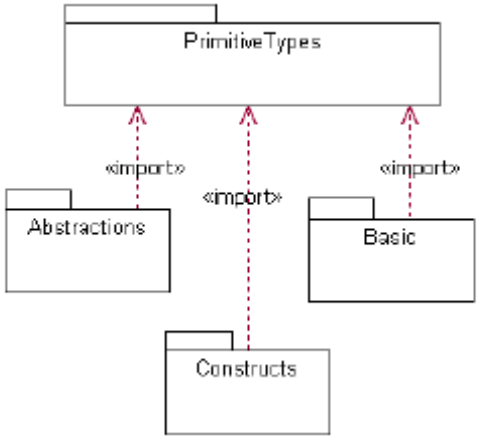

Figure 31: The Core package

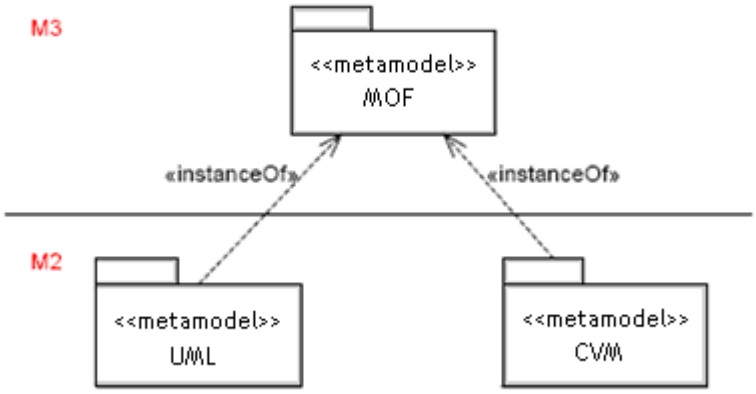

Figure 32: Meta-models relationships

In the case of MOF, the meta-classes of the *InfrastructureLibrary* are used as is, while in the case of UML these model elements are given additional properties. The reason for

these differences is that the requirements when meta-modeling differ slightly from the requirements when modelling applications of a very diverse nature.

4.5 UML Customization

The ability to customize UML to a specific domain is one of the great features of UML. Over the last few years, domain specific modeling languages (DSMLs) have played an increasingly greater role in model-driven development technologies as vehicles for enhancing design productivity and a common approach to DSML design is to use UML as the basic language.

Creating customizations allows one to leverage existing modeling tools and conventions defined by the UML specification while making modeling easier for the end user (and possibly less abstract). As the superstructure specification further points out, there are several reasons why one may want to customize a meta-model:

- Give a terminology that is adapted to a particular domain;
- Give a syntax for constructs that do not have a notation;
- Give a different notation for already existing symbols;
- Add semantics that is left unspecified in the meta-model;
- Add semantics that do not exist in the meta-model;
- Add constraints that restrict the way you use the meta-model;
- Add information that can be used when transforming a model to another model or code.

The type of customizations required often depends on the nature of the domain and how the extended model is intended to be used.

There are three primary methods for defining a DSML, two of which are based on reusing an existing language [17]:

1. *Refinement of an existing more general modeling language* by specializing some of its general constructs to represent domain specific concepts;
2. *Extension of an existing modeling language by supplementing* it with fresh domain-specific constructs;
3. *Definition of a new modeling language from scratch*.

The former two approaches are both possible when applied to the UML and are respectively called *lightweight* and *heavyweight* extensions.

Lightweight extensions involve using profiles. Profiles can be defined by means of those elements contained in the *Profile* package which are aimed at specializing the semantic

of the elements belonging to a meta-model. The primary target for profiles is UML, but it is possible to use profiles together with any meta-model that is based on (i.e., instantiated from) the common core. A profile must be based on a meta-model such as the UML that it extends, and is not very useful standalone. Profiles have been aligned with the extension mechanism offered by MOF, but provide a more light-weight approach with restrictions that are enforced to ensure that the implementation and usage of profiles should be straightforward and more easily supported by tool vendors.

Profiles should be the first instinctive choice when deciding to extend or customize UML. A profile defines limited extensions to a reference meta-model with the purpose of adapting the meta-model to a specific platform domain. The primary extension construct is the *Stereotype*, which is defined as part of a *Profile*. *Stereotypes* can be used to add: keywords, constraints, images, and properties (tagged values) to model elements.

The profile mechanism is not a first-class mechanism, that is to say, it does not allow for specialization through inheritance of meta-types from the referenced meta-model. Rather, the intention of a profile is to give a straight-forward mechanism for adapting an existing meta-model with constructs that are specific to a particular domain. Each such adaptation hast to be grouped in a profile.

Two new concepts have been introduced into the UML2 2.1 API:

- Static Profile Definition: is the ability to create statically defined profiles. Users now have the option to generate code from their profile and provide implementations for operations and derived features.
- OCL Integration: Users can specify invariant constraints or operation bodies in OCL and have code generated from the expressions entered in the UML model. Validation of constraints created on stereotypes is possible after the stereotype has been applied.

Newly introduced in UML 2.0 is also the notion of “strict” application of a profile. This is a means of specifying the kinds of meta-types that your DSL is concerned with. For example, say you were really only interested in Classes, Properties and Operations but you only wanted to specify a stereotype for Class. You could create meta-class reference to Property and Operation. Then when applying your profile, you could specify a “strict” application. UML editors should respect the strict attribute of the profile application and remove all other UML concepts from palettes etc and just leave those that your DSL is concerned with. This is a feature that may or may not be

supported by a given tool. Without a strict application of a profile, all other UML concepts would be available.

Heavyweight extensions are aimed at extending a chosen meta-model with elements and relationships expressed by means of the meta-meta modeling language in which the meta-model is described. Heavyweight extensions of the UML meta-model thus consist of the definition of those elements and relationships representing a certain domain by means of the MOF language thus creating a new domain modeling language which can encompass or extend the UML itself.

With heavyweight extensions it is possible to access to advanced concepts such as sub-setting and redefinition that are used to create UML itself.

The downside to heavyweight extension is that it is the most costly approach being the most difficult to develop. Moreover when developing a new DSML that does not relate to a standard meta-model, such as the UML, several issues may arise.

For example:

- the need to develop a proper editor of the new modeling language that can be exploitable by its end users;
- the need to develop instrument for models interchange.

In order to address these issues several frameworks have been developed in the last years which are aimed at automatically generating editors and other facilities for a DSML starting from the DSML meta-model specification. In the next section we provide a brief description of one of these technologies: the Eclipse Modeling Framework which is the one we have chosen to develop an editor of a domain specific modeling language for context awareness.

4.6 Eclipse and the Eclipse Modeling Framework (EMF)

Eclipse is an open source community, whose projects are focused on building an extensible development platform, runtimes and application frameworks for building, deploying and managing software across the entire software lifecycle. Eclipse began as an IBM Canada project. It was developed by OTI (Object Technology International) as Java based replacement for the Smalltalk based VisualAge family of IDE products, which itself had been developed by OTI. In November 2001 a consortium was formed to further the development of Eclipse as open source. In January 2004 the Eclipse Foundation was created. Eclipse employs plug-ins in order to provide all of its functionality on top of (and including) the runtime system, in contrast to some other applications where

functionality is typically hard coded. Plugins are the key to the seamless integration (but with high interoperability) of tools with Eclipse. With the exception of a small run-time kernel, everything in Eclipse is a plugin. This means that every developed plugin integrates with Eclipse in exactly the same way as other plugins; in this respect, all features are created equal. Eclipse provides plugins for a wide variety of features, some of which are through third parties using both free and commercial models. Examples of plugins include UML plugins for Sequence and other UML diagrams, plugins for DB Exploring, etc.

The Eclipse Modeling Framework (EMF) [20] is an Eclipse's plugin which defines a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a meta-model specification described in a proprietary MOF implementation called ECore, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

While EMF uses XMI (XML Metadata Interchange) as its canonical form of a model serialization, there are several ways of getting a model into that form:

- Create the XMI document directly, using an XML or text editor
- Export the XMI document from a modeling tool such as Rational Rose
- Annotate Java interfaces with model properties
- Use XML Schema to describe the form of a serialization of the model

Once an EMF meta-model has been specified, the EMF generator can create a corresponding set of Java classes which can possibly be edited to add methods and instance variables and still regenerated from the model as needed. The EMF Generator is that software entity which takes as input a meta-model defined throughout the ECore elements and generates:

- A Java implementation of the specified meta-model. The code generated is organized into two Java package: one containing the set of Java interfaces that represent the client interface to the model, the other containing the related implementation. A third optional utility package can be created containing a switch class and an adapter factory, which can be used to attach adapters to the model classes. In general, an interface and an implementation class are generated for each class in the model;

- The Java source code of an Eclipse plugin which realizes an editor for the models which are instances of the specified meta-model. The generated source code consists of two packages named named EMF.edit and EMF.editor. Both these packages extends and builds on the core framework, adding support for generating adapter classes that enable viewing and command-based (undoable) editing of a model, and even a basic working model editor.

Hence, the obtained generated code is actually an eclipse plugin which implements an editor to model instances of the source meta-model. However the generated editor has not to be considered a finished product. In fact attempting to generate an interactive application is fairly tricky undertaking. In generating a data model implementation, the problem is well defined and well contained: to be useful the implementation must correctly and efficiently realize the semantics of the model. Its external interfaces need to be well understood by the developer who whishes to use it as a component in the application being built. By comparison, the problem that EMF.Edit tries to solve is much more open-ended. The code that it generates is meant to be a starting point for the development of an application. Its modification, not just its use, will be necessary. Thus, EMF.Edit must suggest a design and provide utilities that integrate well with its environment while offering enough flexibility that it can be modeled into the developer's desired application.

Building an application using EMF provides several other benefits like model change notification, persistence support including default XMI and schema-based XML serialization, a framework for model validation, and a very efficient reflective API for manipulating EMF objects generically. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

4.6.1 Ecore

EMF can be thought of as a highly efficient Java implementation of a core subset of the MOF model (section 4.3). In other words MOF-like core meta-model in EMF is called ECore. In the current proposal for MOF 2.0, a similar subset of the MOF model, which it calls EMOF (Essential MOF), is separated out. There are small, mostly naming differences between Ecore and EMOF; however, EMF can transparently read and write serializations of EMOF. The Ecore components are related according to the hierarchy depicted in Figure 33 which consists of an extended subset of the taxonomy depicted in Figure 29.

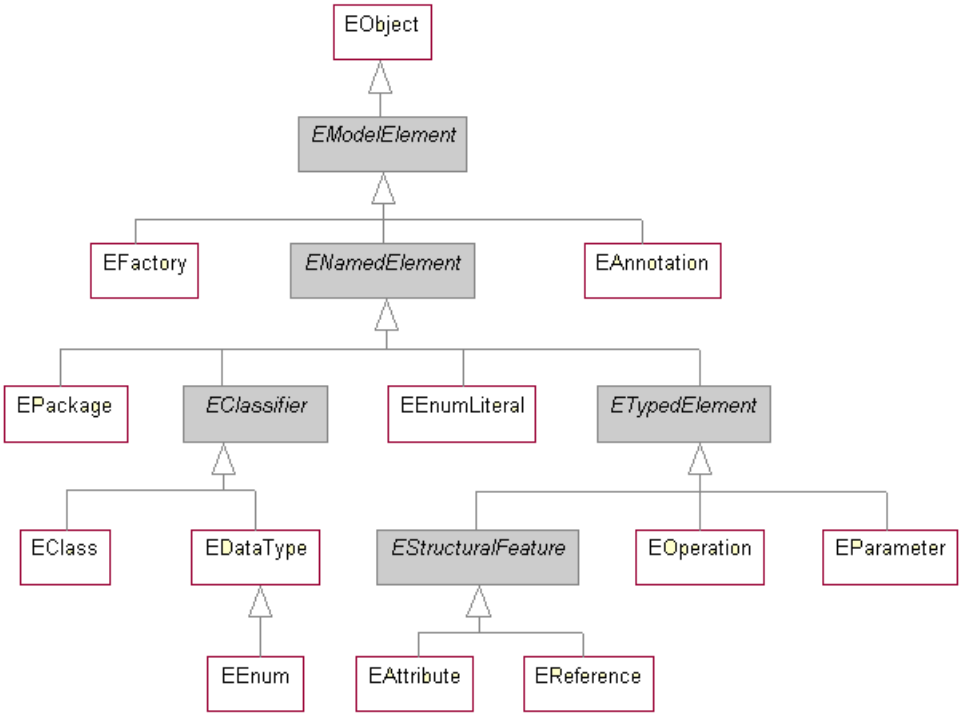

Figure 33: Ecore components hierarchy

Figure 34 thus depicts the kernel element of the ECore model. It essentially defines four types of objects that is, four classes:

- **EClass**: it is a subtype of the EClassifier (the MOF::Classifier equivalent concept) which models classes themselves. Classes are identified by name and can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its supertypes;
- **EAttribute**: models attributes which consist of the components of an object's data. They are identified by name and have a type;
- **EDataType**: models the types of attributes, representing primitive and object data types that are defined in Java, but not in EMF. Data types are also identified by name;
- **EReference** is used in modeling associations between classes; it models one end of such an association. Like attributes, references are identified by name and have a type. However this type must be the EClass at the other end of the association. If the association is navigable in the opposite direction, there will be another corresponding reference. A reference specifies whether to enforce containment semantics;

Because of the similarities between the *EAttribute* and *ERrefence* components (they both have names and types, and taken together, they define the state of an instance of

the EClass that contains them) Ecore includes a common base for them, called *EStructuralFeature* (Figure 33). As depicted in Figure 35, *EStructuralFeature* is, itself, derived from other supertypes such as *ENamedElement* which represents the *MOF::NamedElement* concept thus defining just one name attribute. Most classes in ECore extend this class in order to inherit this attribute.

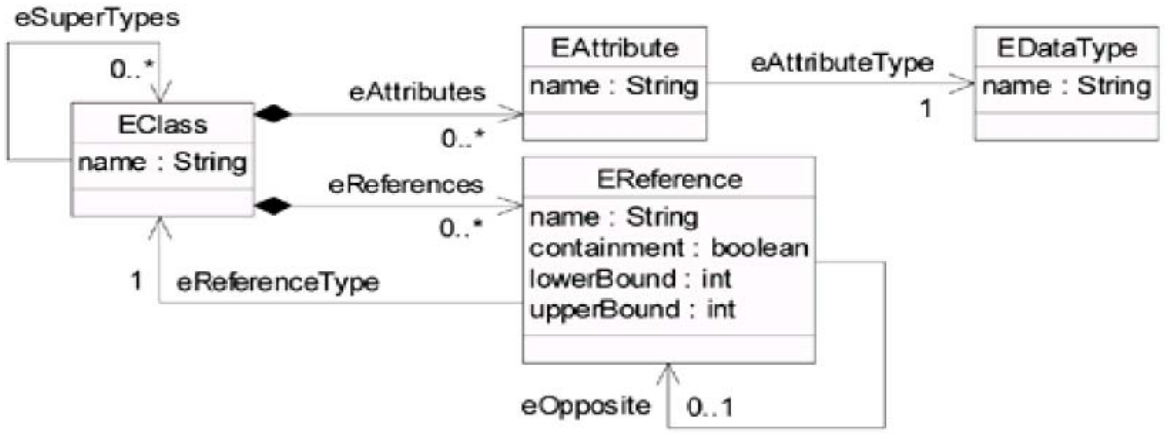

Figure 34: The Ecore kernel

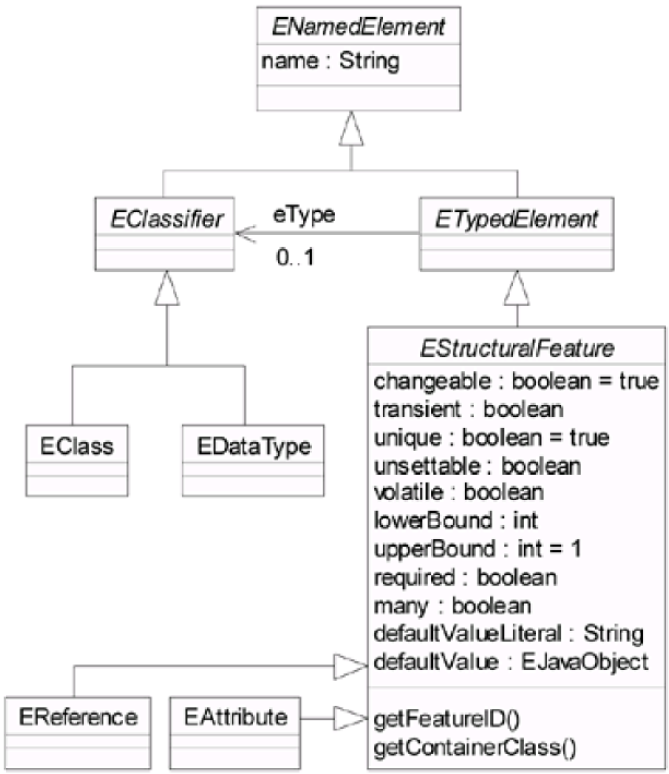

Figure 35: ECore Structural Feature

Another common aspect of *EAttribute* and *EReference* is the notion of a type. Because this is also shared with other classes, in Ecore, the *eType* attribute (Figure 35) is

factored out into *ETypedElement*, the immediate supertype of *EStructuralFeature*. Notice that the type of *eType* is *EClassifier*, a common base class of *EDataType* and *EClass*, which were the required types for *eAttributeType* and *eFeaturedType* respectively.

EStructuralFeature includes a number of attributes used to characterize both attributes and references. Five Boolean attributes define how structural feature stores and accesses values:

- **Changeable**: determines whether the value of the feature may be externally set;
- **Transient**: determines whether the value of the feature is omitted from the serialization of the object to which it belongs;
- **Unique**: which is only meaningful for multiplicy-many features, specifies whether a single value is prevented from occurring more than once in the feature;
- **Unseattable**: specifies whether the feature has an additional possible value, called unset that is unique from any of its type's legal values, including null for an object type. The value of this attribute also determines the semantics of EObject's eUnset() and elsSet() reflective APIs: these methods change an unseattable feature's value to that unset value and test whether it is set to some other value, respectively. But, for a non-unseattable feature, they reset the feature's value to its default and test that it is set to a non-default value, respectively.
- **Volatile**: specifies whether the feature has no storage directly associated with it; this is generally the case when the feature's value is derived purely from the values of other features.

There are four attributes of *EStructuralFeature* that relate to multiplicity, or the number of values associated with the feature. The minimum and maximum number of allowed values are specified by *lowerBound* and *upperBound*, respectively. The last two attributes of *EStructuralFeature* have to do with the default value, the value of the feature before it is explicitly set to something else.

In addition to their structural features, ECore can model the behavioral features of an *EClass* as *EOperation*. Actually, it's really only modeling the interfaces to those operations; a core model gives no further clue as to the actual behavior that operations exhibit. Figure 36 illustrates the *EOperation* class along with the closely related *EParameter*. Notice that *EOperation*'s relationship with *EClass* is quite similar to those of *EAttribute* and *EReference* so that *EOperation*s are contained by an *EClass* via the

eOperations reference, and a derived *eAllOperations* reference is defined to include the operations of a class and its supertypes. The *eOperations* reference is part of a bidirectional association, which allows an *EOperation* to easily obtain the *EClass* that contains it via the opposite reference, *eContainingClass*. An *EOperation* shall contain zero or more *EParameters*, accessible via *eParameters*, which model the operation's input parameters.

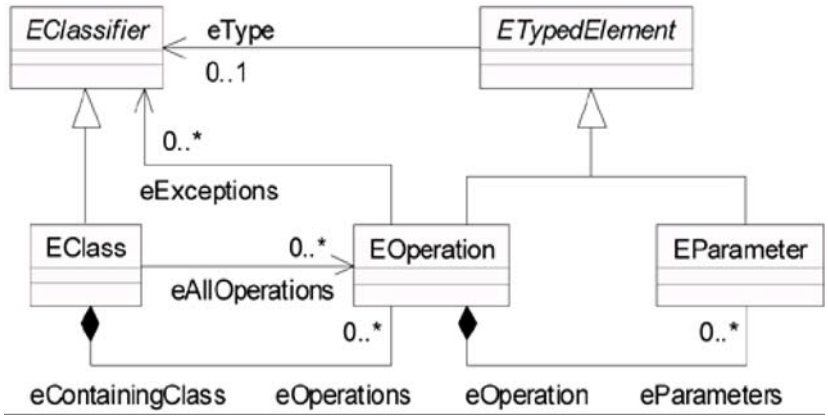

Figure 36: ECore Operations and Parameters

Again, this reference constitutes half of a bidirectional association; the *EParameters* can access the *EOperation* to which they belong via *eOperation*. Both *EOperation* and *EParameter* inherit the name attribute and *eType* reference from *ETypedElement*. These *eType* references model the operation's return type and the input parameter type, respectively, and can refer to any *EClassifier*, whether *EClass* or *EDataType*.
Finally, *EOperation* defines an additional reference, *eExceptions*, to zero or more *EClassifier*, in order to model the types of objects that an operation can throw as exceptions.

4.6.2 The UML2 plugin and the UML2 ECore representation

The Model Development Tool (MDT) [**Errore. L'origine riferimento non è stata trovata.**] project consists of a set of Eclipse's plugins whose purpose is twofold:

- To provide an implementation of industry standard metamodels;
- To provide exemplary tools for developing models based on those metamodels.

The UML2 is itself an Eclipse plugin, belonging to the MDT project, which consists of an EMF-based implementation of the Unified Modeling Language 2.x OMG meta-model for the Eclipse platform.
The objectives of the UML2 component are to provide:

- a useable implementation of the UML meta-model to support the development of modeling tools;
- a common XMI schema to facilitate interchange of semantic models;
- test cases as menas of validating the specification;
- validation rules as means of defining and enforcing level of compliance.

5 An Integrated Framework for Context Oriented Modeling

In this chapter we introduce a framework for context awareness modeling and development which exploits a domain specific modeling language we have appositely designed, as an implementation of the the conceptual domain model described in the 3rd chapter.

In order to easily achieve this objective we have exploited the Eclipse platform and the Eclipse Modeling Framework described in the chapter above. We remind the EMF is an Eclipse's plugin defining a modeling environment and code generation facility for building tools and other applications based on a structured data model. From a meta-model specification formally described in a proprietary meta-meta-modeling language (called ECore), EMF provides tools and runtime support to produce a set of Java classes that enable viewing and command-based editing of the model.

In the following sections we depict how the conceptual model described in the 3rd chapter have been represented using the ECore elements in order to define a domain specific modeling language for context-awareness. Doing this we have assumed the target system, into which context awareness shall be introduced, is modeled by means of the UML [74]. We have therefore extended the UML meta-model with a set of elements aimed at representing the context-aware characteristics of a system. We call this new, domain specific, modeling language CAMEL (Context Awareness ModEling language). From the CAMEL meta-model a proper model editor has been automatically generated exploiting the Eclipse Modeling Framework. In order to fully integrate CAMEL into the Eclipse modeling environment we have also exploited the UML2 Ecore package, described in the paragraph 4.6.2, in order to directly express relationships between elements of a CAMEL model and elements of UML models of the target system.

In this way, because the UML2 ECore implementation is shared with the other UML editors of the Eclipse platform, the CAMEL-models defined through the CAMEL editor can easily refer to the elements of externally defined UML models.

As the CAMEL meta-model represents a concrete instantiation of the conceptual elements introduced in the 3rd chapter, its entities together with their semantics and structures, look very similar to their conceptual counterparts. However, several adjustments have been done in order to gracefully integrate the ECore representation of the conceptual model for context awareness with the ECore representation of the UML 2.0 meta-model.

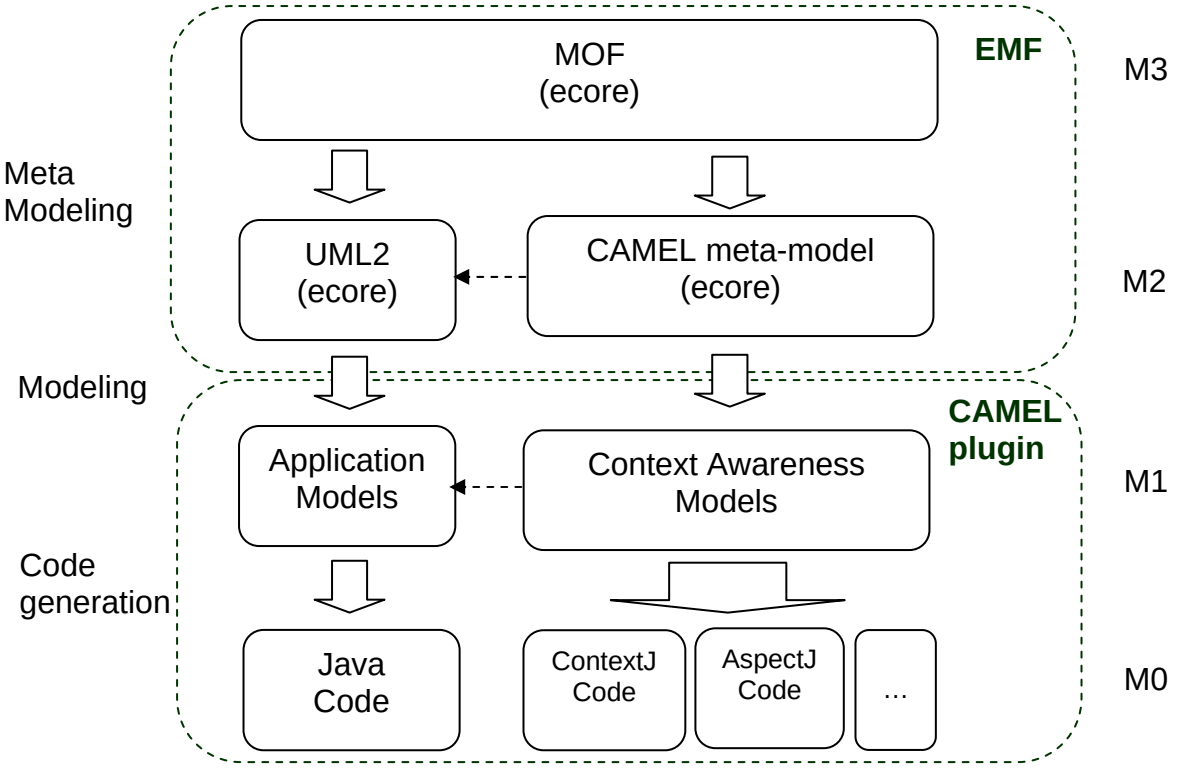

Figure 37: The CAMEL plugin Infrastructure

Therefore, in the following sections, for each element of the CAMEL meta-model, a table summarizing the semantic, structure, and differences with the related conceptual counterpart is provided.

Figure 37 depicts the CAMEL infrastructure as an instantiation of the OMG's modeling architecture depicted in Figure 24. It defines a context oriented MDD process which consists of three phases. The first one consists of the meta-modeling of the CAMEL meta-model by means of the ECore language (M2). This first step is driven by the developers of the CAMEL plugin, exploiting the Eclipse Modeling Framework, and it is therefore invisible to the end user which will only manage the CAMEL model instances. The main output of this phase consists of the CAMEL meta-model and consequently the CAMEL editor itself which can be automatically generated by the meta-model definition. In the second phase, the obtained CAMEL editor plugin is executed into an Eclipse platform by the end user who can exploit it to create and edit new model instances of the CAMEL language (M1). These models can finally be used, in a third phase, as input of a code generation workflow to automatically get their specific realization for a given platform (AspectJ, AspectWerkz, pure Java, C++, etc.)(M0) or other artifacts such as documentation or metrics. It is important to note that CAMEL models, as the UML models, are platform independent so that it is possible to generate the related platform

specific models for any platform to which a proper model transformation rule can be defined.

5.1 CAMEL architecture

Following the main organization of the conceptual model for context awareness, the ECore CAMEL meta-model is defined by means of a set of classes aimed at classifying the remaining concepts into three main categories:

- those related to the modeling of context-data;
- those related to the modeling of context-triggering activities;
- those related to the context-aware adaptation modeling.

The related ECore class diagram representation is depicted in Figure 39 where the *ContextModel* EClass represents an instance of context model and consists of a set of classes called *Contexts*, *Monitors* and *Adapters* which are respectively containers for a set of classes called *Context*, *Monitor* and *Adapter* each one representing the related homonym concept. These classes, together with their semantic attributes and relationships are briefly summarized in the table depicted in figure Figure 38.

EEntity	**Semantics**	
ContextModel	EClass which represents a model for context awareness. It is the container for all the other classes defined in the CAMEL language and can have a unique instance with respect to a model instance.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. Each context model is characterized by a name.
	contexts	A context model is a container of contexts by means of an aggregation with the *Contexts* EClass.
	monitors	A context model is a container of monitors by means of an aggregation with the *Monitors* EClass.
	adapters	A context model is a container of adapters by means of an aggregation with the *Adapters* EClass.
EEntity	**Semantics**	
Contexts	EClass which acts as container for the contexts of a context model.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class.

		Each Context is uniquely identified by a name with respect to the context model to which it belongs.
	contexts	The *Contexts* EClass has the functional role to act as container *Context's* instances by the means of an aggregation with the *Context* class (paragraph 5.3).
EEntity	**Semantics**	
Monitors	EClass which acts as container for the monitors of a context model.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. Each monitor is uniquely identified by a name with respect to the context model to which it belongs.
	monitors	The *Monitors* EClass has the functional role to act as container for *Monitor*'s instances by means of an aggregation with the *Monitor* class (paragraph 5.3).
EEntity	**Semantics**	
Adapters	EClass which acts as container the adapters of a context model	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. Each Adapter is uniquely identified by a name with respect to the context model to which it belongs.
	adapters	The *Adapters* EClass has the functional role to act as container for *Adapter*'s instances by means of an aggregation with the *Adapter* class (paragraph 5.4).

Figure 38: CAMEL essential concepts

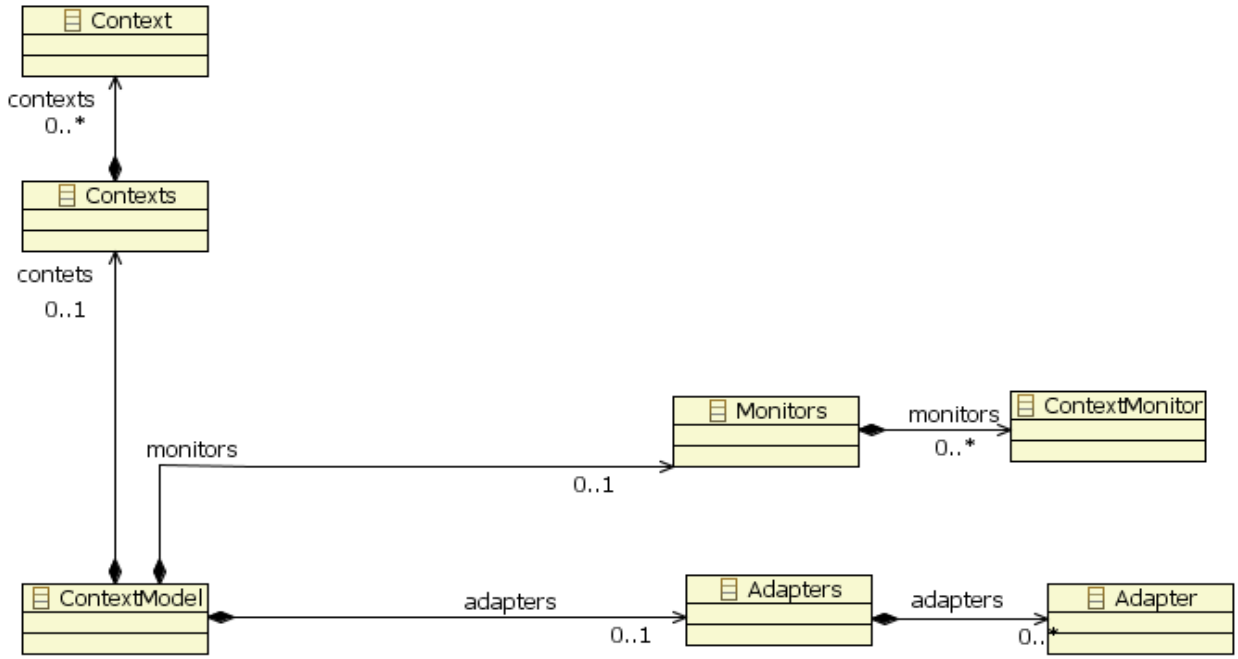

Figure 39: The CAMEL Conceptual Architecture

5.2 The Modeling of Contextual Data in CAMEL

Figure 40 depicts the ECore diagram representing concepts used to define the model of context data in CAMEL. This diagram is very similar to its conceptual counterpart (Figure 8). State based and event based contexts are defined. These are the ECore instantiation of the two possible ways of representing contextual information: respectively as a set of key-value pairs or a set of events. Both these kind of contexts are modeled by Ecore Classes (EClass) thus they are *ENamedElement* and inherit a structural feature representing their name.

Also in the CAMEL language, a state based context consists of a set of attributes, represented by the *ContextAttribute* concept (itself an *EClass*), that are supposed to be relevant for a given context entity. In order to relate to target system model expressed by means of the UML, in the CAMEL language the context attribute semantic has been restricted so that the source of a context attribute consists of an explicit relationship with a *UML::TypedElement.* The UML infrastructure [74] defines a *TypedElement* as an "*element that has a type that serves as a constraint on the range of values the element can represent"*. More precisely it is a concept that may represents structural features such as: attributes, operations, operations parameters, operations return parameter, associations, references, etc. The reference association between a context attribute and *a UML::TypedElement* means that a context attribute may not own the UML

TypedElement representing its source but it is supposed to be externally defined, usually in a UML model of the target system.

CAMEL context attributes may instead own *UML::Operations* which can be appositely defined to provide contextual information indirectly related to the context attribute's source. These operations can for example encapsulate some sort of elaboration of the source data whose returning value can then be referred as source by other context attributes.

The *EventBasedContext* consists of a set of events, represented by the *ContextEvent* concept, that are supposed to be relevant for a context entity. As for the context attribute, in order to relate to target system model expressed by means of the UML, the semantic given to the *ContextEvent* concept has been restricted in the CAMEL language to the set of events representable by a *UML::Operation*. The UML superstructure [74] defines an *Operation* as "*a behavioural feature of a classifier that specifies the name, type, parameters and constraints for invoking an associated behavior"*. In the CAMEL modeling environment, relating a context event to a UML::Operation means to associate that event to a well identifiable behaviour of the target system.

The CAMEL *Context* may also be built as an aggregate of other contexts, both state-based and event-based. This is possible through the *CompositeContext* EClass which is the third type of *Context*, itself implemented by an EClass, whose unique objective is to be a container for logically related State and Event based contexts.

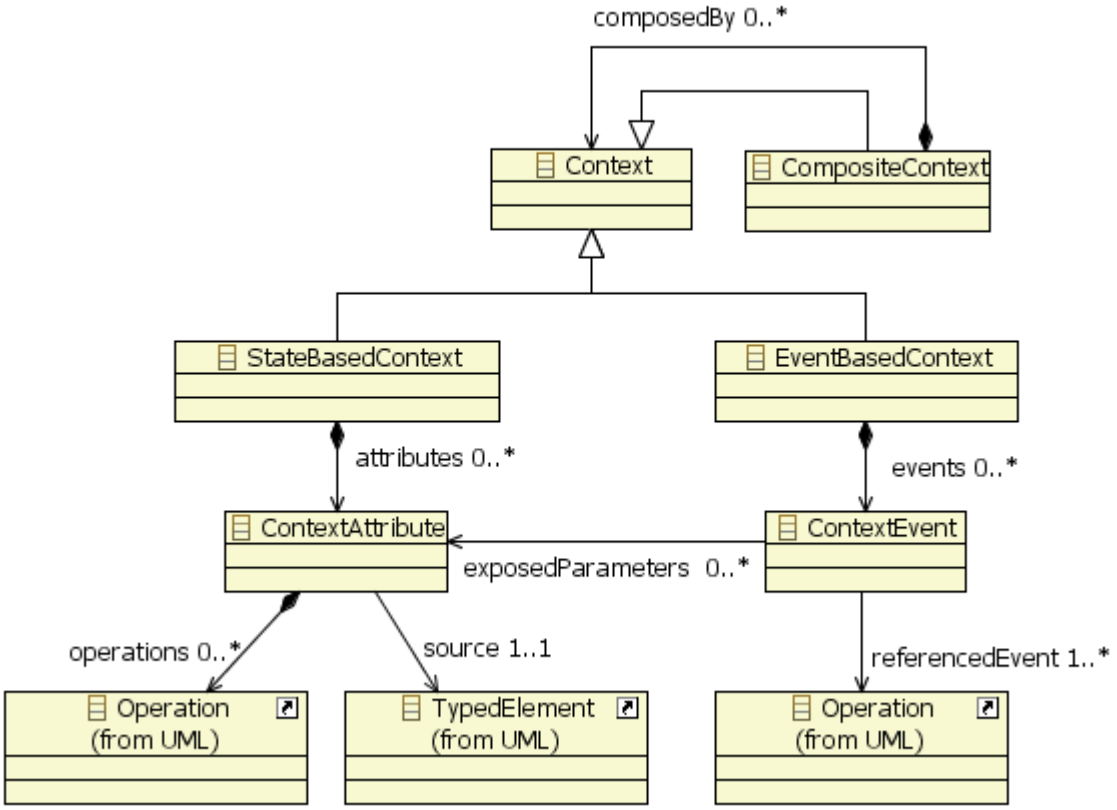

Figure 40: The CAMEL Context data modelling concepts

EEntity	**Semantics**	
Context	EClass representing the *Context* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class it uniquely identifies a context within a context model.
EEntity	**Semantics**	
CompositeContext	EClass representing the *CompositeContext* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	Name	Inherited from the Context class with same semantic.
	composedBy	A composite context is a special context that can be composed by other contexts by means of an aggregation, called *composedBy*, with the *Context* class. It can be therefore considered a container for logically related contexts both state based or event based.
EEntity	**Semantics**	
StateBasedContext	EClass representing the *StateBasedContext* concept. It has the	

	same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the Context class with the same semantic.
	attributes	A state based context is a container of context attributes. By means of an aggregation named *attributes* with the *ContextAttribute* class.
EEntity	**Semantics**	
ContextAttribute	EClass representing the *ContextAttribute* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. It uniquely identifies a context attribute within the state based context to which it belongs.
	source	Association which specifies by which entities the context attribute takes its value. In CAMEL this association is realized by a reference with the *UML::TypedElement* concept. Therefore, in CAMEL models, a context attribute can takes its value by any typed element belonging to any UML model.
	operations	In CAMEL a context attribute can also has associated operations which model some sort of elaboration over its *source*. These operations can be defined when the source data has to be properly elaborated before it can be exploitable. This *ContextAttribute's* characetic is modeled by means of an aggregation with the *UML::Operation* class.
EEntity	**Semantics**	
EventBasedContext	EClass representing the *EventBasedContext* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the Context class with the same semantic.
	events	An event based context is a container of *ContextEvents* by the means of an aggregation named *events* with the *ContextEvent* class.
EEntity	**Semantics**	
ContextEvent	EClass representing the *ContextEvent* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**

	name	Inherited from the EClass class. It uniquely identifies a *ContextEvent* within the *EventBasedContext* to which it belongs.
	referencedEvent	Association which specifies to which event the context event is related. In CAMEL this association is realized by the means of a referenced called *referencedEvent* with the *UML::Operation* concept.
	exposedParameters	When a ContextEvent is part of a composite context it can expose some of the values of the context attributes belonging to the same composite context when the context event is fired.

Figure 41: The CAMEL Context data modelling concepts

As an example, consider a very simple prototypical target systems represented by the class diagram depicted in Figure 42. This class diagram has been designed through the Eclipse's UML editor and made persistent by a file named *targetSystem.umlclass*. Imagine now we want to be aware of some contextual information while the system is executing. As an example suppose we are interested in being aware about the current value of the *propertyOne* attribute of the *ClassA's* instances and aware about when the *methodOne* and *methodTwo* methods, respectively belonging to *ClassA* and *ClassB,* instances are invoked. In order to model what described we can exploit the CAMEL editor creating a CAMEL model relating to the independently modelled class diagram. Figure 43 depicts the aforementioned CAMEL model.

In order to refer to the elements of the target system class diagram the *targetSystem.umlclass* has been loaded as an external resource. Conseguently a composite context is created consisting of two coxtexts, respectively state and event based.

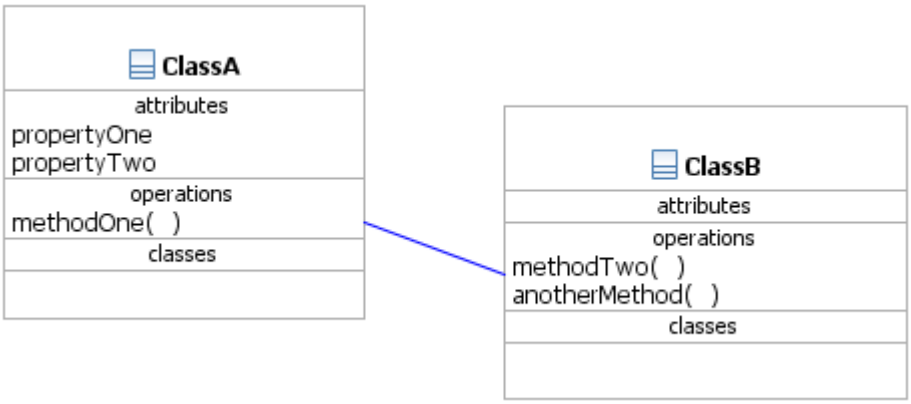

Figure 42: A target system class diagram

The former is composed by a context attribute named *batteryLevel* having as source (Figure 43) *the ClassA::propertyOne* attribute. The latter consists of two context event, namely: eventA and eventB respectively referring to the *ClassA::methodOne()* (Figure 44) and *ClassB::methodTwo().*

It is important to note that the original target system models have been not modified when the model of contextual feature has been introduced.

Similarly to aspect oriented modelling, CAMEL context attributes and context events act as pointcuts with respect to the loaded target system models making possible for the designer to encapsulate in well defined entities (the contexts) references to the context entities from which contextual information has to be retrieved.

Contexts provide an alias for each context entities so that context monitoring and context adaptation models are not coupled to the target system entities and can possibly be reused in the modelling of new context aware systems.

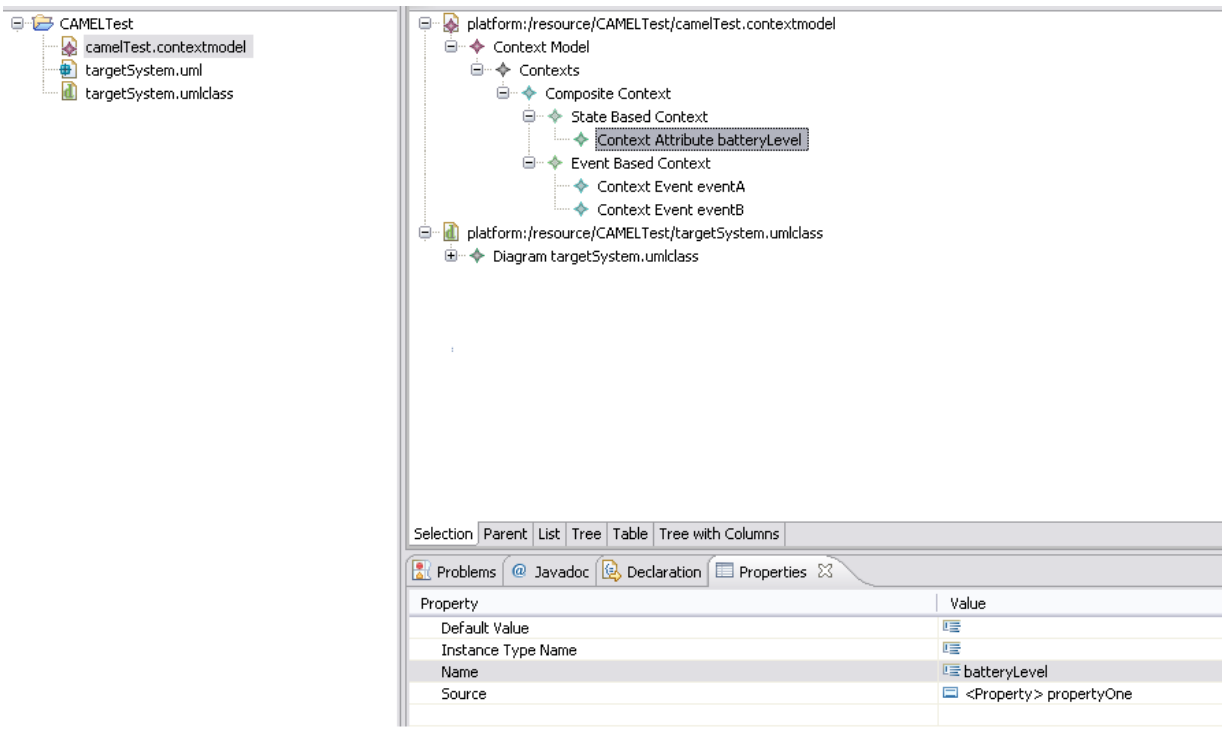

Figure 43: An example of context sensing modelling

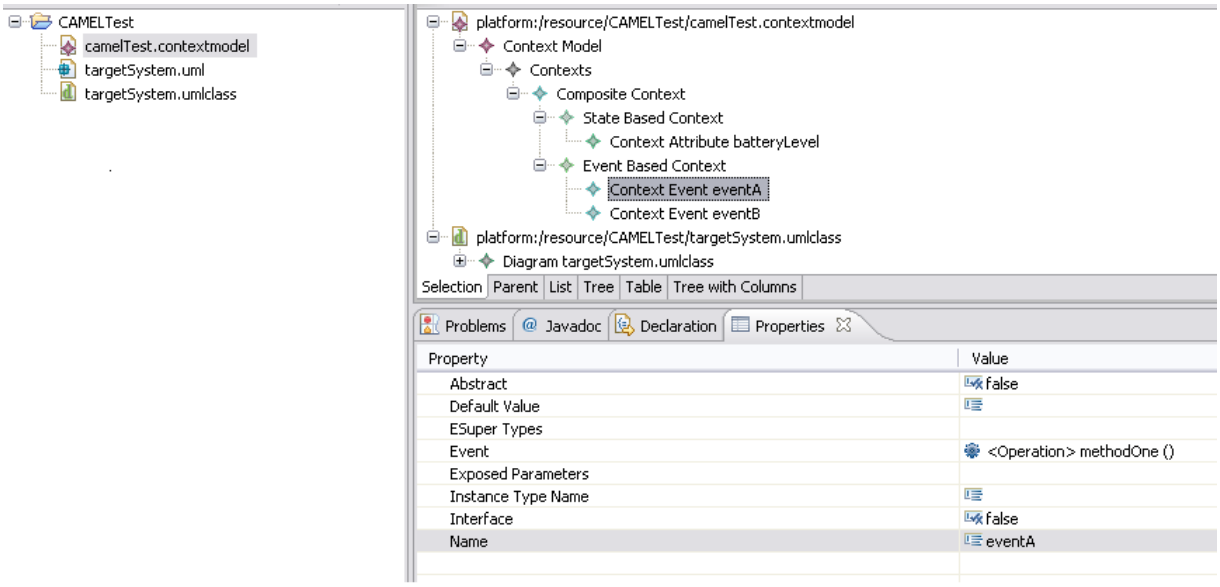

Figure 44: An example of context sensing modelling

5.3 The Modeling of Context Adaptation Triggering in CAMEL

In CAMEL, the modeling of the context adaptation triggering concern is based on a set of modeling elements representing a possible realization of the concepts contained in the CAT conceptual package described in section 3.2.

Figure 45 depicts the ECore representation of these elements while Figure 46 depicts a table in which they are defined by means of their semantics, the semantics of their attributes and the semantics of their associations.

The *ContextMonitor* class is the main container of the CAMEL elements for the modeling of the context monitoring concern. As its conceptual counterpart, it acts as container of logically related context states. A context state represents an interesting condition over the contextual information brought by the contexts the related monitor is actually observing.

Each context state refers to an instance of *ContextCostraint,* abstract *EClass* representing the CAMEL realization of the homonym conceptual element (Figure 11), representing the condition that must hold to consider the related context-state active. In the CAMEL language there can be three kinds of context constraints represented by the *StateConstraint*, *EventConstraint* and *Operator* classes. These three classes extend the *ContextConstraint* abstract class but have different semantics.

The *StateConstraint* is the CAMEL realization of its conceptual homonym counterpart (section 3.2). It represents a logical/mathematical condition over the value of a context attribute to which the constraint is related by means of a reference association. A state constraint also makes possible to specify an alias for the target entity (i.e. an object, etc.) which is source of the context attribute having activated the constraint. This alias, specified by means of the *instanceReference* attribute, can be eventually exploited by adaptation activities triggered by the constraint.

In the CAMEL metamodel the *EventConstraint* is the CAMEL realization of its homonym conceptual counterpart (section 3.2). It is defined by means of a set of events represented by the *ContextEvent* concept. An event constraint also makes possible to specify an alias for the target entity (i.e. an object, etc.) which is source of the context events having activated the constraint. This alias, specified by means of the *instanceReference* attribute, can be eventually exploited by adaptation activities triggered by the constraint.

While the semantics of an event constraint is that it holds if one of its referred context events occurs, the state constraint never holds on its own but dependently on the kind of the *Operator* which are associated to it.

The *Operator* class is the third kind of *ContextConstraint* provided by the CAMEL language, it represents a composite constraint through the definition of a condition making a set of state or event based constraints verified. An operator constraint consists

of two attributes: *function* and *value*; and an aggregation association with the *ContextConstraint* class itself.

The CAMEL language provides two different ways of defining an operator. The former approach (section 5.3.1) is the most expressive and consists of defining as *function* attribute the condition that the operator actually represents over the context constraints it contains. This approach is very expressive because no assumption is done about the syntax the designer can adopt to define such a condition. In this way the designer is free to choose the grammar which is most suitable for his/her modeling purposes. Because the syntax describing how these conditions have to be specified is not part of the CAMEL meta-model this approach has the drawback they can not be processed in standard CAMEL model transformations. It is up to the designer to extend or implement new transformations that take into account the adopted syntax for condition specifications.

The latter approach (section 5.3.2) consists of exploiting the *Operator* class and the aggregation relationships between it and the *ContextConstraint* class in order to specify pattern of events as state machines or logical/mathematical conditions over state based contexts. This approach is less expressive than the previous one because pattern of events can be expressed only by means of type-3 grammars (regular expressions or state machines). However because the syntax describing how these conditions have to be specified is part of the CAMEL meta-model, it is very simple to eploit already existing or implement new transformation rules able to interpret them.

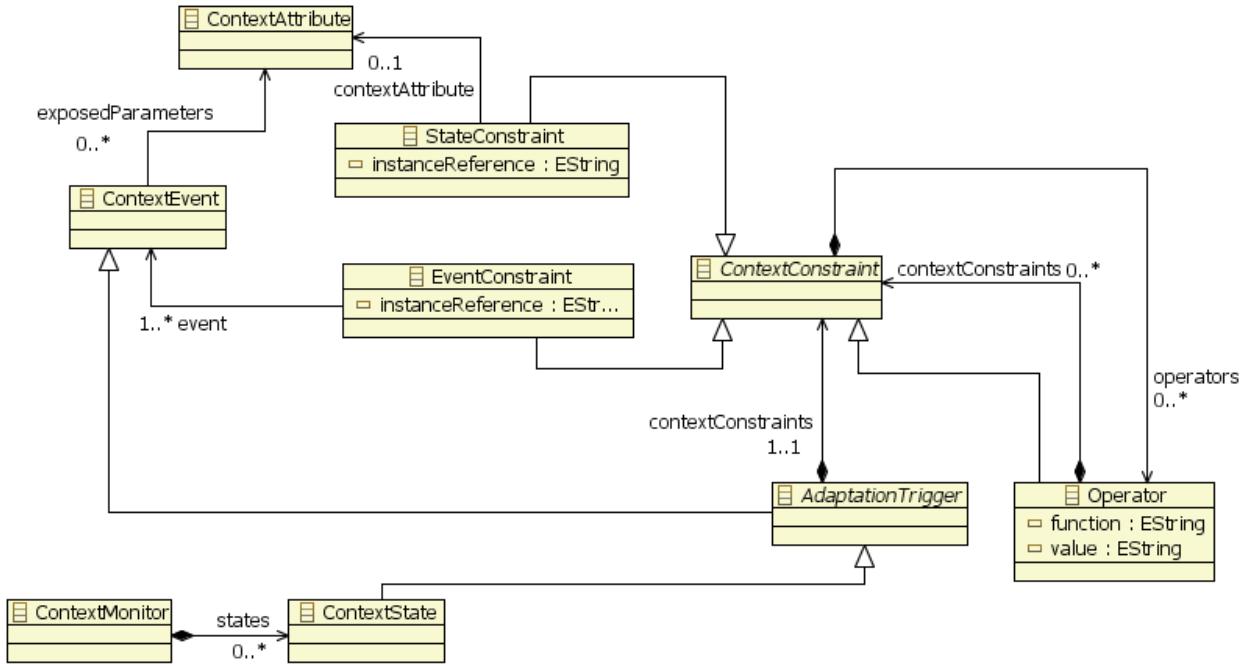

Figure 45: The CAMEL Context Adaptation Triggering EClasses

EEntity	**Semantics**	
ContextMonitor	EClass representing the Monitor concept. It has the same semantic of the homonym conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. It uniquely identifies a monitor within a context model.
	states	A context monitor is a container of context states by means of an aggregation named *states* with the *ContextState* class.
EEntity	**Semantics**	
AdaptationTrigger	EClass representing the *AdaptationTrigger* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. It uniquely identifies an adaptation trigger within a context model
EEntity	**Semantics**	
ContextConstraint	EClass representing the *ContextConstraint* concept. It has the same semantics of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the *AdaptationTrigger* class with the same semantic.
	operators	Each context constraint can have

		associated several operators. Operators can be exploited to define composite constraints, logical/mathematical condition over a state constraints or sequential constraints over event constraints.
EEntity	**Semantics**	
StateConstraint	EClass representing a restricted semantic of the *StateConstraint* concept. As its conceptual counterpart, in the CAMEL, a state constraint represents a logical/mathematical predicate on the value of a context attribute. In the CAMEL this is realized throughout the operators to which the *StateConstraint* has to be associated.	
	Attributes & Associations	**Semantics**
	name	Inherited from the *AdaptationTrigger* class with the same semantic.
	operators	Inherited from the *AdaptationTrigger* class. In a state constraint the aggregated operators can be exploited both to define a logical/mathematical predicate over the context attribute, to which the state constraint relates, or to compose a state constraint with other constraints both state or event based.
	instanceReference	Alias representing a reference to the target entity source of the context attribute having activated the constraint.
	contextAttribute	Reference association with the context attributes to which the state constraint refers.
EEntity	**Semantics**	
EventConstraint	EClass representing a restricted semantic of the *EventConstraint* concept. As its conceptual counterpart, in the CAMEL, an *EventConstraint* is defined by means of a precise pattern of events that can either be *ContextEvents* or other *AdaptationTriggers*. However, in the CAMEL, each *EventConstraint* on its own is defined by a set of events/triggers so that it holds as soon as one of them occurs. More complex patterns can be defined through the use of operators.	
	Attributes & Associations	**Semantics**
	name	Inherited from the AdaptationTrigger class with the same semantic.
	operators	Inerhited from the *AdaptationTrigger* class. The operators can be exploited to define complex pattern of events

		throughout a proper composition of the container event constraint with other event constraints each one representing a set of events that have to respect the sequential rules imposed by the operators.
	instanceReference	Alias representing a reference to the target entity source of the context event having activated the constraint
	events	Reference association with the context events to which the state constraint refers. If no operators are defined the constraint is verified as soon as one of its referred events occurs.
EEntity	**Semantics**	
ContextState	EClass representing the *ContextState* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited by the *AdaptationTrigger* class with the same semantic.
	contextConstraint	A context state goes active as soon as the contextConstraint it owns is satisfied.
	exposedParameters	When fired, a context state can expose the value of some of those context attributes which are part of the context information that has satisfied its contextConstraint.

Figure 46: The CAMEL Context Adaptation Triggering EClasses

5.3.1 Operator generic syntax

As aforementioned the CAMEL language leaves the designer free to adopt the grammar for expressing constrainning conditions which is most suitable for his/her modeling purposes. In this case the designer can exploit the operator's *function* attribute to specify contraints by means of an arbitrary language.

Because the syntax by which such conditions are specified is not part of the CAMEL meta-model this approach has the drawback they can not be processed in standard CAMEL model transformations. It is up to the designer to extend or implement new transformations that take into account the adopted syntax for condition specifications.

Figure 47 depicts the semantics of the CAMEL *Operator* when it is used with the generic syntax approach.

<table>
<tr><th>EEntity</th><th colspan="2">Semantics</th></tr>
<tr><td>Operator</td><td colspan="2">EClass representing an context constraint operator.
It consists of two attributes: function and value; an aggregation association with the ContextConstraint and an aggregation with the Operator class itself.
The function attribute represents the actual condition an Operator instance actually represents over the context constraints it contain. No assumptions are done about the syntax by which this condition has to be specified.</td></tr>
<tr><td></td><td>Attributes & Associations</td><td>Semantics</td></tr>
<tr><td></td><td>name</td><td>Inherited from the ContextConstraint class.</td></tr>
<tr><td></td><td>contextConstraints</td><td>Aggregation with the ContextConstraint class which makes an operator container of ContextConstraints.</td></tr>
<tr><td></td><td>operators</td><td>Inherited from the ContextConstraint class it makes an operator container of other Operators.</td></tr>
<tr><td></td><td>function</td><td>Represents the actual semantic of the operator. When the left operand is a StateConstraint it can assume the following values:</td></tr>
<tr><td></td><td>value</td><td>Not used</td></tr>
</table>

Figure 47: The CAMEL Constraint Operator with generic syntax and semantics

Figure 48 and Figure 49 depict two simple examples of how the constraint operator with generic syntax can be exploited to model context constraints. In the former example (Figure 48) a *ContextState* named *lowEnergy* is modeled as constrained by an *Operator* named *batteryLow* containing a state constraint named *batteryLevel* which is associate to a context attribute representing a variable in a target system where the battery state is constantly updated. The *batteryLow* operator automatically activates the *lowEnergy* context state as soon as the value contained in the batteryLevel state constraint is lesser than 0,4.

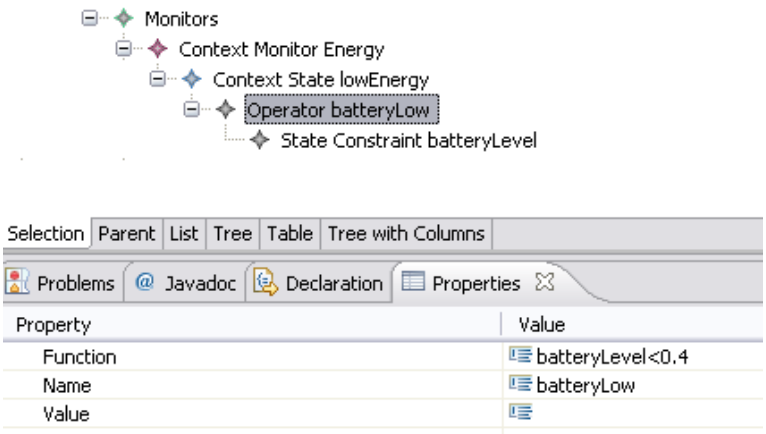

Figure 48: Example of context constraing with generic syntax

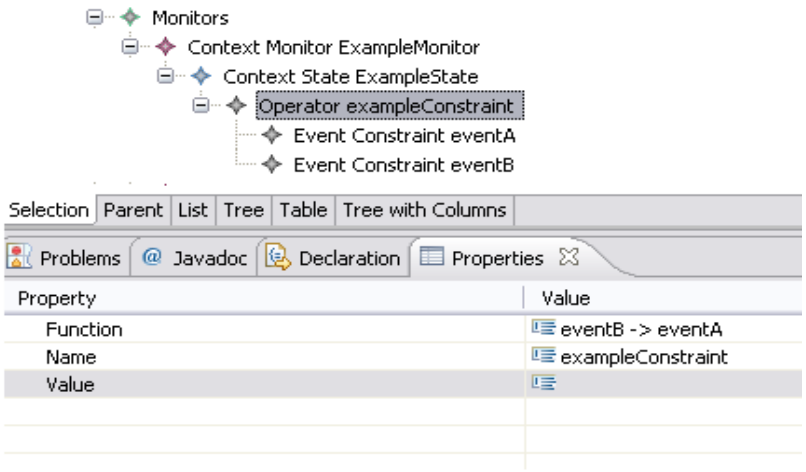

Figure 49: Example of context constraing with generic syntax

In the second example (Figure 49) a context state named *ExampleState* is modeled as constrained by an operator named *exampleConstraint* containing two event constraints respectively representing the occurrence of a context event named *eventA* and a context event named *eventB*. The *exampleConstraint* operator automatically activates the *exampleState* context state as soon as the *eventA* occurs after the *eventB*. This condition is specified in the *function* attribute of the *exampleConstraint* operator by means of an arbitraty language.

5.3.2 Operator CAMEL syntax

The CAMEL meta-model also provides a built-in way of defining constraining conditions exploiting the Operator class in order to specify pattern of events as state machines or logical/mathematical conditions over state based contexts. This approach forces the designer to adopt the CAMEL syntax but can very reduce the complexity of interpreting specified constraining conditions during model transformations.

When this approach is adopted, the *function* attribute represents an atomic function of an *Operator* instance. The context constraint containing the operator has to be considered the left operand of the function specified by the operator. The right operand can be either represented by the *value* attribute or by another context constraint contained by the operator itself.

An operator's function can be: a mathematical operator (i.e. =, <, =<, >, >=, etc.) when the containing constraint is a state constraint; a sequential specification (i.e., AFTER, BEFORE, etc.) when exploited to compose event constraints; a logical operator (i.e. AND, OR, NOT) when used to compose contexts both state and event based. The *function* attribute has to be necessarily specified while the *value* attribute is facultative to the kind of *function* the operator represents. It is meaningful only when the *function* is a logical or mathematical operator. In this case the whole constraint holds if the context attribute referred by the state constraint which contains the operator verifies the operator's function with respect to the specified value. If the operator contains another *StateConstraint* the *value* attribute is not meaningful and the right operand of the operator's function is represented by the value of the context attribute referred by the state constraint contained by the operator.

EEntity	**Semantic**
Operator	EClass representing a composite constraint through the definition of a condition making a set of state or event based constraints verified. It consists of two attributes: *function* and *value*; an aggregation association with the *ContextConstraint* and an aggregation with the *Operator* class itself. The *function* attribute represents the actual function of the operator. The context constraint which contains the operator has to be considered the left operand of the function specified by the operator. The right operand can be either represented by the value attribute or by another context constraint if present. An operator's function can be: a mathematical operator (i.e. =, <, =<, >, >=, etc.) when the containing constraint is a state constraint; a temporal specification (i.e., AFTER, BEFORE, etc.)

	when exploited to compose event constraints; a logical operator (i.e. AND, OR, NOT) when used to compose contexts both state and event based. The *function* attribute has to be necessarily specified while the *value* attribute is facultative to the kind of *function* the operator represents. It is meaningful only when the *function* is a mathematical operator. In this case the whole constraint holds if the context attribute referred by the state constraint which contains the operator verifies the operator's function with respect to the specified value. If the operator contains another state constraint the *value* attribute is not meaningful and the right operand of the operator's function is represented by the value of the context attribute referred by the state constraint contained by the operator. Operators which are at the root of the constraint that contains them have to be considered in an AND relation.		
	Attributes & Associations	**Semantics**	
	name	Inherited from the *ContextConstraint* class.	
	contextConstraints	Aggregation with the *ContextConstraint* class which makes an operator container of context constraints. If present, the contained constraints represent the right operand with respect to the function specified by the operator (where the constraint containing the operator is the left operand). Because of this characteristic it is mutually exclusive with the *operators* attribute.	
	operators	Inherited from the *ContextConstraint* class it makes an operator container of other *Operators*. If present, the contained operators represent the right operand with respect to the function specified by the containing operator. Because of this characteristic it is mutually exclusive with the *contextConstraints* aggregation.	
	function	Represents the actual semantic of the operator. When the left operand is a StateConstraint it can assume the following values:	
		=	The value of the context attribute referred by the State constraint which contains the operator has to be equal than the value specified in the value attribute or the value of the

			context attribute referred by the state constraint contained by the operators
		>	The value of the context attribute referred by the State constraint which contains the operator has to be greater than the value specified in the value attribute or the value of the context attribute referred by the state constraint contained by the operators
		<	The value of the context attribute referred by the State constraint which contains the operator has to be lesser than the value specified in the value attribute or the value of the context attribute referred by the state constraint contained by the operators
		When exploited to compose context constraints it can assume the following values	
		AFTER	When the operator function takes this value it has to container an *EventConstraint*. The semantic is that the whole constraint holds if the left operand of the operator's function has been verified after its right operand.
		BEFORE	When the operator function takes this value it has to container an *EventConstraint*. The semantic is that the whole constraint holds if the left operand of the operator's function has been verified before its right operand.
		OCCURRENCES	The operator's function can takes this value only if the left operand is an *EventConstraint*. The semantic is that the whole constraint holds as soon as the context event referred

			by the event constraint which contains the operator has occurred the number of times specified in the value attribute.
		AND	The whole constraint is verified if the contained constraints are all verified.
		OR	The whole constraint is verified if on of the contained constraints is verified.
		NOT	The whole constraint is verified if the contained constraint is not verified.

Figure 50: The CAMEL Constraint Operator with the CAMEL syntax and semantics

Figure 51-a depicts an example of state constraint in the CAMEL language. It shows the model of a monitor named *Energy* which is supposed to be the observer of the target system's energy level. It consists of two context-state named *highEnergy* and *lowEnergy*. In the figure, the latter context state is expanded so that it is possible to note that it consists of a state constraint referring to a context attribute named *batteryLevel*, itself referring to a structural element of the target system which is supposed to store the data about the platform energy. An operator is associated to such state constraint having function '<' and value '0.4'. The whole constraint holds when the value of the platform energy goes below the value 0.4 thus activating the related context state named *lowEnergy*. An alternative modelization of the same scenario is depicted in Figure 51-b. In this case the whole constraint holds whenever the value of the context attribute referred by the context constraint named *batteryLevel* is lower than the value of the context attribute referred by the context constraint named *someOtherValue* .

As aforementioned, when applied to event constraints, operators can be used to model pattern of events. As mentioned in section 3.2, there can be several ways to define pattern of events whose expressiveness may strongly vary depending on the kind of grammar that has been chosen to produce these patterns (Figure 12). In the CAMEL language, pattern of events can be modeled according to a type-3 grammar which is the kind representing finite-state automata and regular expression. The rules defining this grammar are described in the table depicted in Figure 50. In particular, when operators are used to model pattern of events, their *function* attribute can also assume the values AFTER or OCCURRENCES.

Figure 52 depicts a simple example of how these two operators can be exploited to model pattern of events in CAMEL. The AFTER operator has to be used to detect those patterns in which an context event occurs after another one. Figure 52-a depicts a simple example where a context state named *ExampleState* is activated whenever a context event named *eventA* has occurred after a context event named *eventB*. The OCCURRENCE operator instead, has to be used to detected those pattern in which a *ContextEvent* hast to occur for a specified number of time. Figure 52-b depicts a simple example where the same context state, *ExampleState*, is activated when the *eventA* context event has occurred for the third time. In this case the value attribute of the related operator has to be specified with the number of occurrences the related context event must have. Jolly characters can be used such as '+' which states the event has to be occurred at least 1 time, or '*' which states the event has to be occurred 0 or more times. The logical operator OR can finally be exploited to model alternative flows within a complex pattern of events.

When a context state is fired, as soon as the context constraint it owns is verified, parameters can be passed to the adapters triggered by the activated context-state. These parameters can be selected within the set of those context attributes that are involved in the context definition and are represented by the *exposedParameters* reference which associate the *ContextState* and the *ContextAttribute EClasses*.

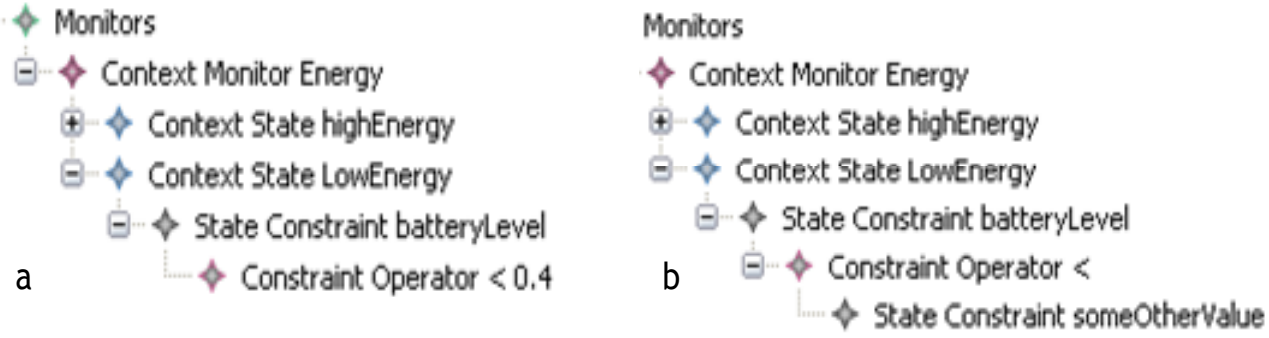

Figure 51: Examples of State Constraint in CAMEL

Figure 52: Example of Event Constraints in CAMEL

5.4 The Modeling of Context Adaptation in CAMEL

In the CAMEL language, the modeling of context adaptation concerns is based on a set of meta-modeling elements representing a realization of those concepts contained in the COA conceptual package described in section 3.3.

Figure 53 depicts the ECore representation of these elements while Figure 54 depicts a table in which they are introduced through a brief description of their semantics, the semantic of their attributes and the semantic of their associations. The main container of the CAMEL elements for the modeling of the context adaptation concern is the *Adapter* Class. As its conceptual counterpart, an adapter (Figure 53) is a container of *AdaptationLayer* instances. An adaptation layer is a container of adaptation mechanisms represented by the *AdaptationMechanism* EClass, an abstract class which generically represents a mechanism exploitable to react to context changes. By means of the *triggeredby* reference, an adaptation layer can be associated to a context state representing the condition that, when verified, activates all the adaptation mechanisms contained by the layer.

In the *CAMEL* language there can be two types of *AdaptationMechanism*: *Insert* and *Binding*. As its conceptual counterpart, a binding represents a pair consisting of an entity and a specified value. When the adaptation layer to which a binding belongs is triggered by a context state the binding goes active and instantaneously substitutes the value of the specified target entities with its own value. A binding is realized by an *EClass* with two references to *UML::NamedElement* respectively named *where* and *what*. The former reference represents the target entity affected by the binding while the latter represents the *UML::NamedElement* whose value will be substituted to the value of the first one as soon as the adaptation layer to which the binding belongs will be triggered by a context state. In the UML superstructure [74] a named element is an

abstract concept representing an element in a model that may have a name. The use of the *UML::NamedElement* concept makes possible for the binding to refer to any element with a name defined in a UML target model. A Binding finally makes possible to specify, by means of the *instanceReference* attribute, an alias to a specific instance of the referred *NamedElement* to which it has to be applied. The actual value of this instance is given by the instance references specified in the state and event constriants having triggered the activation of the binding. If the instance reference attribute is not specified the binding will be applied to all the instances of the *NamedElement* referred by the *where* association. Otherwise only the specified instance will be affected.

The *Insert* class is itself an abstract *AdaptationMechanism*. It represents the introduction of new elements into already existing entities of the target UML model. Similarly to the conceptual model, in the CAMEL language there can be two concrete realization of the *Insert*: *StructuralInsert* and *BehavioralInsert*. The former is aimed at introducing structural features (such as Properties, Parameters, etc.), specified through the *what* aggregation to a set of *UML::StructuralFeature* concept, into already existing entities of the target model, specified by means of a *where* reference to a *UML::Class*. In the UML superstructure, a structural feature is an abstract concept representing a typed feature of a classifier that specifies the structure of instances of the classifier. It is therefore a super class for classes' properties and operation's parameters.

A behavioral insert instead is aimed at introducing behavioral features specified through the *what* aggregation to a set of *UML::BehavioralFeature* concept, defined in the UML superstructure and representing the signature of the behavior to be introduced, into an already existing entity of the target model, specified throught he *where* referehce to a *UML::Class*. A proper model of the introduced behaviors (activity diagrams, state diagrams, etc.) has then to be provided by the introduction of bindings which bind together the desired UML models with the behaviors defined in the related behavioral inserts.

Both structural and behavioral inserts finally make possible to specify, by means of the *instanceReference* attribute, an alias referring to a specific instance of the *Class* to which they have to be applied. In the case of behavioral insert this alias can also be referred by the model of the added behaviours.

The actual value of this instance is given by the instance references specified in the state and event constraints having triggered the activation of the insert. If the instance reference attribute is not specified the insert will be applied to all the instances of the

Class referred by the *where* association. Otherwise only the specified instance will be affected.

Binding and inserts remains active until the related adaptation layer is active. A part of these two steady kinds of adaptation it is also possible to specify in the model of an adaptation layer two mechanisms, namely *incomingActivity* and *outgoingActivity*, which respectively represent an activity to be performed once as soon as the layer is activated and as soon as the layer is deactivated.

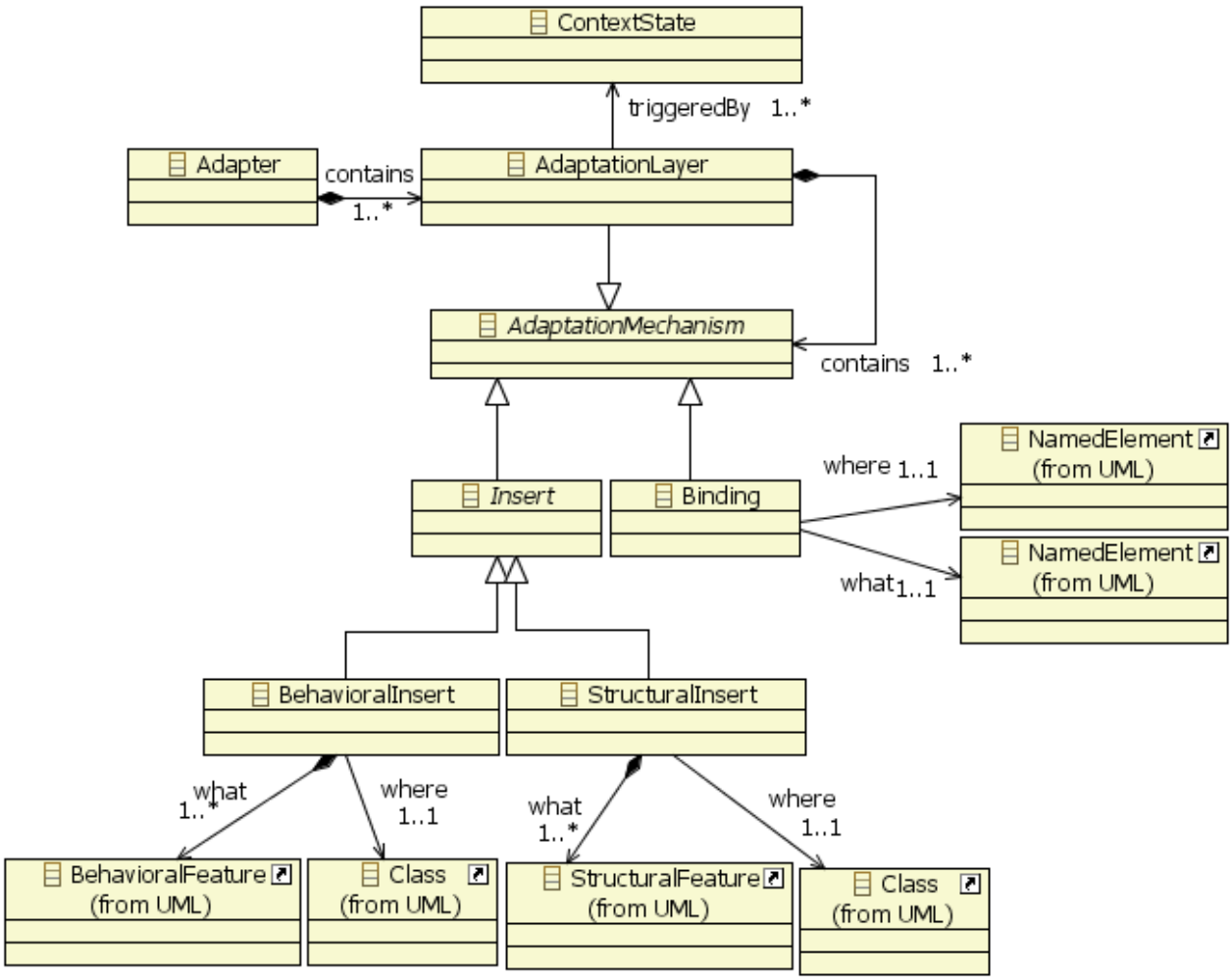

Figure 53: The CAMEL Context Adaptation EClasses

EEntity	**Semantics**	
Adapter	EClass representing the CAMEL realization of the *Adapter* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. It uniquely represents an Adapter with

		respect to a context model.
	contains	As its conceptual counterpart, an adapter is a container of adaptation layers by means of an aggregation association with the *AdaptationLayer* EClass
EEntity	**Semantics**	
AdaptationMechanism	EClass representing the CAMEL realization of the *AdaptationMechanism* concept. It has the same semantics of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class. It uniquely represents the adaptation mechanism with respect to the adapter that contains it.
EEntity	**Semantics**	
AdaptationLayer	EClass representing the CAMEL realization of the *AdaptationLayer* concept. It has the same semantic of its conceptual counterpart.	
	Attributes & Associations	**Semantics**
	name	Inherited from the *AdaptationMecharnism* class with the same semantic.
	triggeredBy	As its conceptual counterpart an adaptation layer is activated as soon as a *ContetConstraint*, to which it refers throughout the *triggeredBy* reference, is verified.
	contains	An adaptation layer is a container of logically related adaptation mechanism such as *Insert* or *Binding* throughout the *contains* aggregation. The semantic is that as soon as the adaptation layer goes active the adaptation mechanism it contains are automatically activated. As a consequence as soon as an adaptation layer is deactivated the contained adaptation mechanisms are automatically deactivated too.
	incomingActivity	It represents activities that have to be performed as soon as the triggering context state is fired and the layer activated.
	outgoingActivity	It represents activities that have to be performed as soon as the layer is going to be deactivated and so the related context state.

<table>
<tr><th>EEntity</th><th colspan="2">Semantics</th></tr>
<tr><td>Binding</td><td colspan="2">EClass representing the CAMEL realization of the Binding concept. As its conceptual counterpart, in the CAMEL language a Binding is defined by a pair consisting of an entity and a value. When a Binding is activated it substitutes the value of the specified target entity with its own value. Depending on the kind of entity this mechanism is applied to, it can be used to achieve different types of adaptation. For example, it can be used to bind a service interface to different implementations, a service invocation to different services, a parameter in a service invocation to different values. The main difference between the CAMEL implementation and is conceptual definition is that, in the CAMEL language, for the sake of simplicity, only one value can be defined for a Binding. That is why there is not a realization of the BindingOperation concept.</td></tr>
<tr><td></td><td>Attributes & Associations</td><td>Semantics</td></tr>
<tr><td></td><td>name</td><td>Inherited from the AdaptationMechanism class.</td></tr>
<tr><td></td><td>instanceReference</td><td>It is an alias representing an instance to which the binding has to be applied. If it is not specified the binding will be applied to all the instances of the NamedElement referred by the where association. Otherwise only the specified instance will be affected.</td></tr>
<tr><td></td><td>where</td><td>In the CAMEL language the entity target of a Binding is a UML::NamedElement that can be referenced throughout the where association. In the UML meta-model a NamedElement is any element in a model that may have a name. These elements can be (Classes, structural feature, behavioral features, operations, etc.)</td></tr>
<tr><td></td><td>what</td><td>The what association specifies the entity that has to be substituted to the one referenced by the where association. Also the what association consists of a reference with a UML::NamedElement.</td></tr>
<tr><th>EEntity</th><th colspan="2">Semantics</th></tr>
<tr><td>Insert</td><td colspan="2">EClass representing the CAMEL realization of the Insert concept. As it conceptual counterpart an insert models the introduction of new structural or behavioral elements into an already existing entity. It consists of a specification of the value that must be inserted, and of the point within some</td></tr>
</table>

	application entity where it must be inserted.	
	Attributes & Associations	**Semantics**
	name	Inherited from the *AdaptationMechanism* class with the same semantic.
EEntity	**Semantic**	
BehavioralInsert	EClass representing the CAMEL realization of the *BehavioralInsert* concept with the same semantic of its conceptual counterpart. In the CAMEL realization of this concept, the *BehavioralPointcut* is represented by a UML Class of the target UML model while the *BehavioralValue* by a *UML::Operation*.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class.
	instanceReference	It is an alias representing an instance to which the insert has to be applied. If it is not specified the insert will be applied to all the instances of the *Class* referred by the *where* association. Otherwise only the specified instance will be affected.
	Where	Reference to the UML::Class representing the *BehavioralPointcut*.
	what	Aggregation of *UML::BehavioralFeatures* representing the insert's *BehavioralValues*.
EEntity	**Semantics**	
StructuralInsert	EClass representing the CAMEL realization of the *StructuralInsert* concept. It has the same semantic of its conceptual counterpart. In the CAMEL realization of this concept the *StructuralPointcut* is represented by a UML Class of the target UML model while the *StructuralValue* is represented by a *UML::StructuralFeature*.	
	Attributes & Associations	**Semantics**
	name	Inherited from the EClass class.
	instanceReference	It is an alias representing an instance to which the insert has to be applied. If it is not specified the insert will be applied to all the instances of the *Class* referred by the *where* association. Otherwise only the specified instance will be affected.
	Where	Reference to the UML::Class

		representing the *StructuralPointcut*.
	what	Aggregation of *UML::StructuralFeatures* representing the insert's *StructuralValues*.

Figure 54: The CAMEL Context Adaptation EClasses

As an example suppose we are interested in introducing a new attribute, called *addedProperty*, and a new method called *addedMethod,* to the ClassA instances (Figure 42) experimenting a given context. Suppose the attribute is an integer typed field and the method takes a double typed parameter and returns a boolean result. In order to model this adaptation in CAMEL, an activity diagram representing the *addedMethod* behavior has to be properly defined by means, as an example, of the Eclipse's UML editor (Figure 55 depicts the modeled addedMethodBehavior activiti diagram).

Figure 56 and Figure 57 depict the CAMEL model of the designed adapter, called *test adapter*. It consists of a layer named *testLayer* containing a structural insert named *testInsert* and a behavioral insert named *testInsert2* both having the pointcut referenced to the *ClassA* class. When activated the insert introduces an integer typed property called *addedProperty* to the involved instaces of this class *(*Figure 56).

The modeled behavioral insert instead introduce a new operation named *addedMethod* to the involved instances of ClassA. The signature of this method is modeled in order to take a double typed parameter and returns a boolean value.

The behavioral insert only specifies the signature the added operations will have. In order to specify also the behavior such signature represents, a binding has to be defined. To this end, the *addedMethodBinding* (Figure 57) links the *newBehavior* activity diagram, which has been loaded in the CAMEL model as external resource, to the new added behavior introduced by the *testInsert2* inser.

This simple example shows how the CAMEL language can be exploited in order to design the way independently defined UML entities have to be dynamically woven to properly handel contextual information.

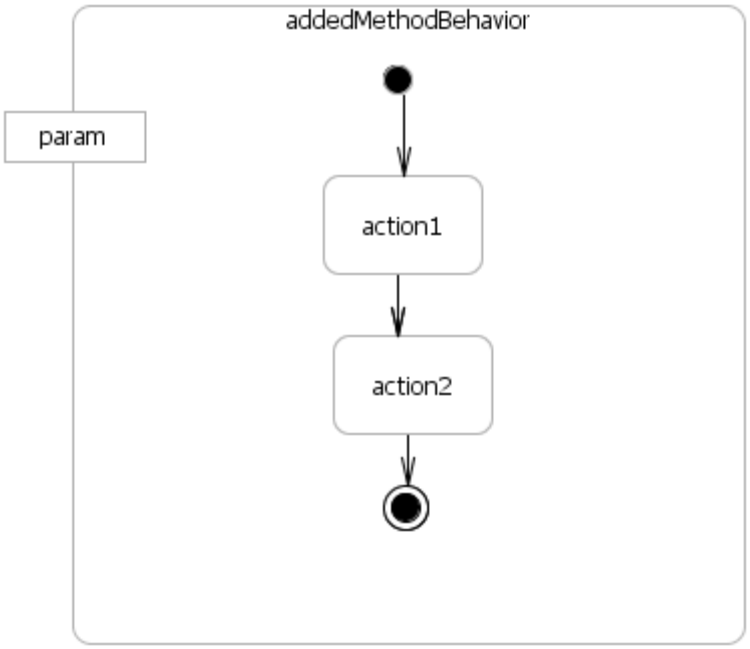

Figure 55: A model for the addedMethod behavioural insert

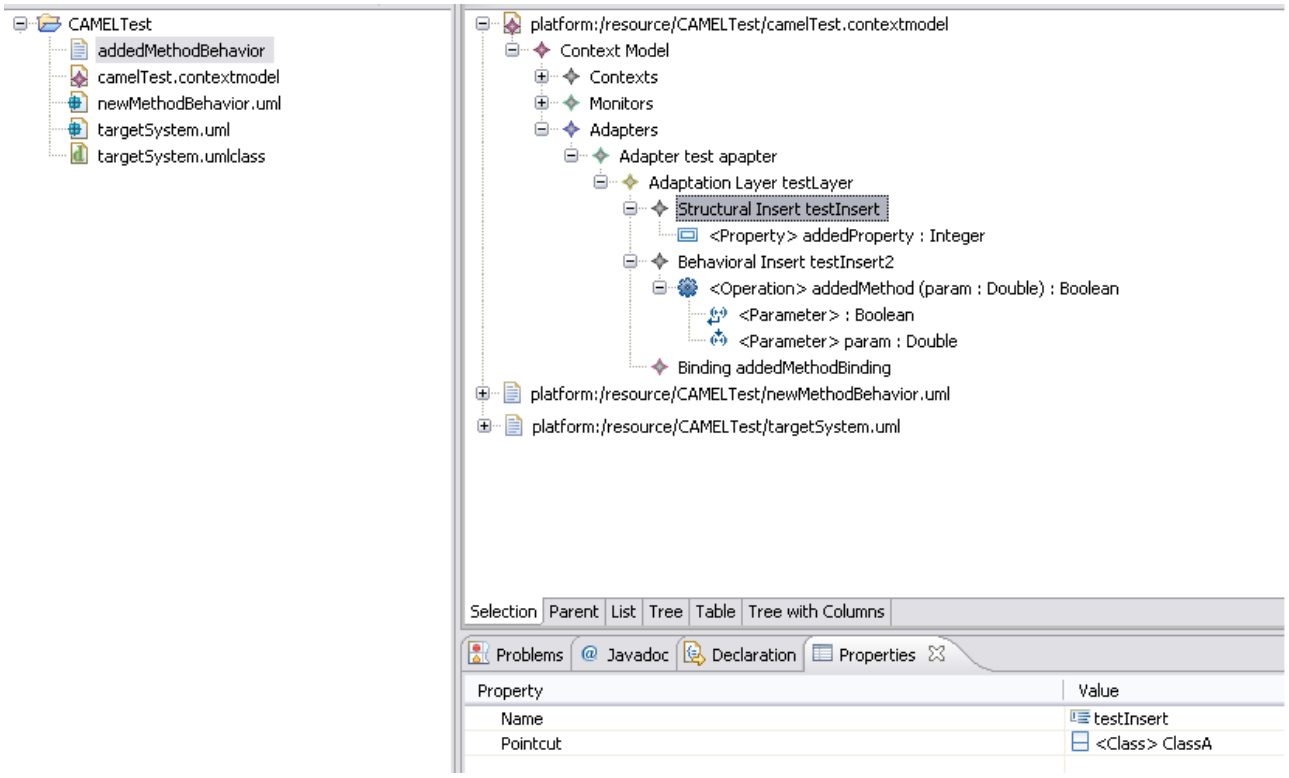

Figure 56: An example of CAMEL adapter and adaptation layer

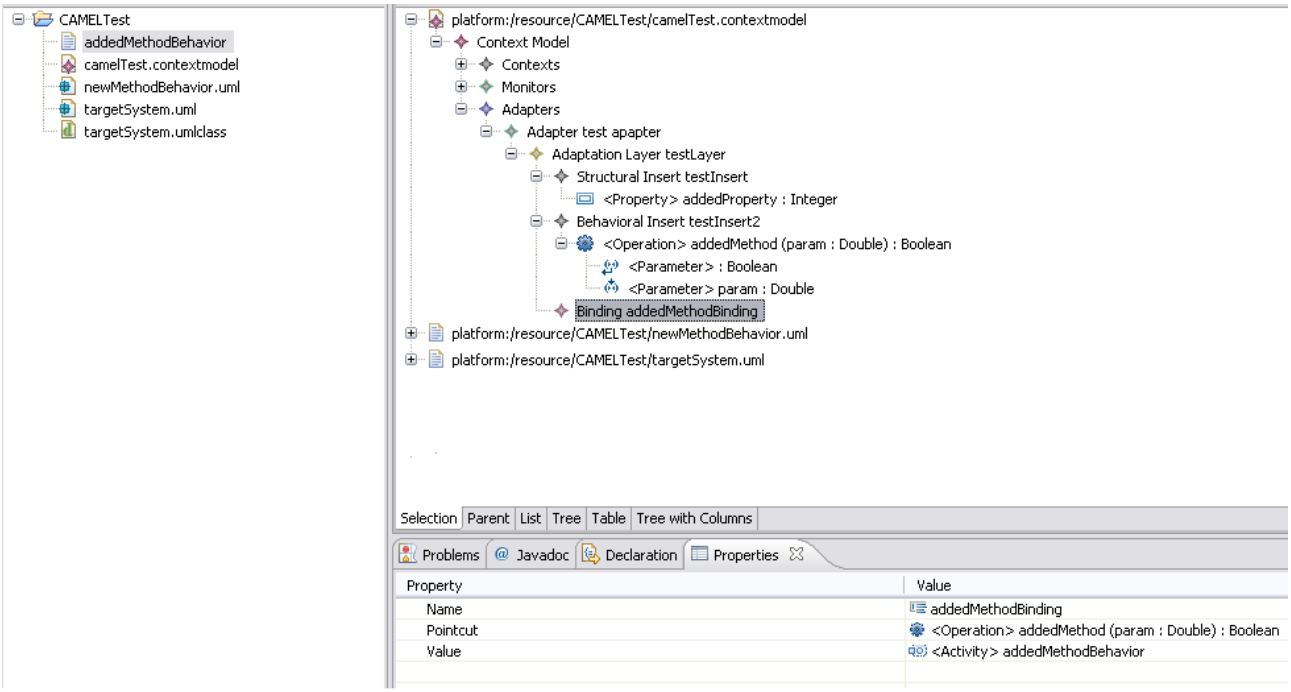

Figure 57: An example of CAMEL adapter and adaptation layer

6 JCOOL a Java Context Oriented Language for Context Oriented Development

In the Model Driven Development paradigm, code is just a model having the characteristic of being executable with respect to an execution platform. Because of this characteristic it can either be considered a platform specific model or even platform independent when the final target platform requires other model transformations in order to get an executable representation of it.

As an example, imagine a platform independent model defined by means of the UML meta-model. From this model a Java representation may be produced through proper vertical/horizontal exogenous transformations. The obtained Java code, or its related byte-code, can then be considered the final PSM with respect to the Java Virtual Machine (JVM) but it is platform independent with respect to the operating system on the top of which the JVM is actually executed. It is the JVM itself that, during the system execution, automatically transforms the platform independent *bytecode* into its platform specific representation.

CAMEL models can be considered platform independent models with respect to an execution platform. They in fact consist of an abstract representation of structural and behavioral characteristics a target system should provide to properly exploit contextual information and possibly react to context changes. These models can act as input to several model transformations aimed at producing different artifacts such as: performance metrics; documentation; etc. As a consequence code is just only one of the possible artifacts that can be derived from CAMEL models but is probably the one the designer is most interested in.

Because of the crosscutting nature of context awareness, two major approaches can be employed during such transformation. The former consists of the generation of source code in which the modeled context-aware characteristics have been woven with the code representing the system business logic. This approach is typically legitimated with respect to a MDD process considering that once the designer can automatically generate code from models he/she will have no need to preserve separation of concerns at code level but will probably prefer to focus on performances.

The separation of concerns can be otherwise preserved even at code level through the generation of aspect oriented source code implementing the context aware

characteristics and object oriented code implementing the business logic of the designed system.

Both these two approaches are applicable with our framework. The models built with the CAMEL plugin are in fact independent of the kind of code that can eventually be generated from them. It is up to the designer to choose the transformation that is most suitable for his/her purposes.

However we believe separation of concerns at code level should be preserved in order to increase the code understandability that may be needed when dealing with agile approaches requiring developers to directly operate over the generated source code.

As mentioned in the 2nd section, *Context Oriented Development* techniques represent the best way to develop systems context aware characteristics. Several technological platforms have already been defined to address context oriented development issues where *ContextToolkit* and *ContextJ* are probably the most widely adopted. As briefly described in paragraphs 2.3.1 and 2.3.2, these two languages address two distinct complementary tasks: the former is aimed at defining a distributed architecture of context widgets which are elements aimed at providing contextual information, retrieved by the underlying context-sensors, to those entities which are interested in; the latter by the contrary is aimed at introducing adaptation mechanisms by means of layers of operations that can be activated by client applications in order to change the behaviors of those components which are affected by the layers. We can therefore state the *ContextToolkit* addresses the concerns of context sensing and context monitoring, even for distributed systems, while *ContextJ* addreses the context adaptation concern providing mechanisms to dynamically activate and deactivate layers of alternative context dependent behaviors. These two complementary approaches lack a good application of the separation of concerns principle with respect to the target application to which they tend to be too much invasive. *ContextJ* Layers have in fact to be defined within the system classes (Figure 4) so that the original code of the target system has to be strongly modified in order to introduce context awareness features. Moreover Layers implementation is not well encapsulated into a single component but is scattered throughout the classes it involves. The same considerations hold for code needed to activate/deactivate layers that has to be directly introduced within the business logic code. Also the Context Toolkit requires the developer to spread the code related to the context sensing concern throughout the components implementing other concerns, eventhough these information are then encapsulated into the widget constructs.

Inspired by these languages, which have introduced new promising ways to develop context oriented systems, but aimed by the intention to reduce the coupling between a target application and the code implementing context aware characteristics, we have started to define a new context oriented programming language, named JCOOL (Java COntext Oriented Language) that tries to address both these issues.

JCOOL (Java Context Oriented Language) is a Java extension that has been explicitly designed for the context oriented programming keeping in mind the separation of concerns principle that has to be preserved between the context awareness concerns and the other concerns composing a system. JCOOL consists of a code level implementation of the concepts introduced in the conceptual framework described in the 3rd chapter and has been presented in [21] for the first time but then consistently modified in order to increase the separation of concerns between: context definition, context adaptation triggering and context adaptation activities.

Because of what argued, one of the main characteristic of JCOOL is that it gives to developers the possibility to extend already existing Java applications with context aware capabilities without affecting the original Java code. Similarly to ContextJ, JCOOL provides means to define and activate layers of context dependent behaviors that substitute specified default behaviors of a given target system. The main difference is that, in JCOOL, both these concerns are well encapsulated into separated components which are not tangled with the code implementing the other concerns of the target system.

6.1 JCOOL Syntax and Semantic with ANTLR

JCOOL introduces four first class constructs to the Java language: *Context*, *Monitor*, *Adapter* and *AdaptationLayer* which are very similar to the homonym concepts defined in the conceptual framework for context awareness depicted in the 3rd chapter. In this section, we provide a description of the syntax and semantics related to each of these constructs by means of an ANTLR grammar [101,102] that has been developed for the JCOOL parser. ANTLR (Another Tool for Language Recognition) is a language tool that provides a framework for constructing recognizers, interpreters, compilers and translators from grammatical descriptions containing actions in a variety of target languages. It provides support for tree construction, tree walking, translation, error recovery and error reporting. In order to have a prototypical JCOOL parser that should automatically verify the syntactical correctness of a JCOOL program and automatically

generate the target code implementing it, we have exploited ANTLR thus defining a grammar representing the whole valid JCOOL sentences.

A Grammar describes the syntax of a language. In other terms we may say a grammar generates a language because grammars are usually exploited to describe what languages look like. However the goal of formally describing a language is tipically aimed at obtaining a program that recognizes sentences in the language generated by the grammar. With respect to this objective, ANTLR is a program that automatically converts grammars to such recognizers. A grammar is defined by means of a set of rules each one describing some phrase (subsentence) of the language. The rule where parsing begins is called the start rule and each rule consists of one or more alternatives.

The most common grammar notation is called Backus-Naur Form (BNF) [103]. ANTLR uses a grammar dialect derived from YACC [100] where rules begin with a lowercase letter and token types begin with an uppercase letter. For the sake of room we do not describes the ANTLR syntax of its grammar dialect but refer to the ANTLR documentation [102] for a complete description of it.

Figure 58 depicts the JCOOL Lexicon that is to say its vocabulary. It consists of: the special terms *Context*, *Monitor, Adapter, AdaptationLayer, involves* and *observes* together with logical operators and brackets; white space characters (WS); new line special character (NS) and *identifier*. With the *identifier* token we refer to those sequence of characters which are characterized by zero or more alpha-numerical characters together with the symbols '<', '>' and '*'.

Figure 61 depicts the rules which are common to the JCOOL's constructs, aimed at modeling the syntactical structure of methods and attributes declaration. We have identified a way to refer to existing method which is represented by the *method* rule and a way to declare new method which is identified by the *methodDeclaration* rule.

```
tokens {
        MONITOR = 'Monitor';
        CONTEXT = 'Context';
        INVOLVES = 'involves';
        OBSERVES = 'observes';
        ADAPTER  = 'Adapter';
        LAYER    = Layer;
        AND = ',';
        OR = '||';
        NOT = '!';
        OPEN = '(';
        CLOSE = ')';
}
WS : (' '|'\t'|'\n'|'\r')+ {skip();};
NEWLINE : '\r'? '\n' {skip();};
Identifier : ('a'..'z'|'A'..'Z'|'0'..'9'|'>'|'<'|'*' )*;
```


Figure 58: JCOOL Lexicon

The former rule, represented by the syntax diagram depicted in Figure 59, states a method reference has to start with a type, can be followed by a Class name and possibly by an instance specification, has to be followed by the method name and possibly by a sequence of parameters. The *methodDeclaration* rule, represented by the syntax diagram depicted in Figure 60, states a method definition has to start with a type followed by a method name possibly followed by a sequence of parameters after the parameters a body has to be specified wrapped by curly brackets.

The *attribute* rule specifies the syntactical structure that has to be followed when referencing to an existing attribute in the JCOOL language. With respect to this rule, an attribute reference has to start with a type specification possibly followed by a class and instance specification and finally by the attribute name itself.

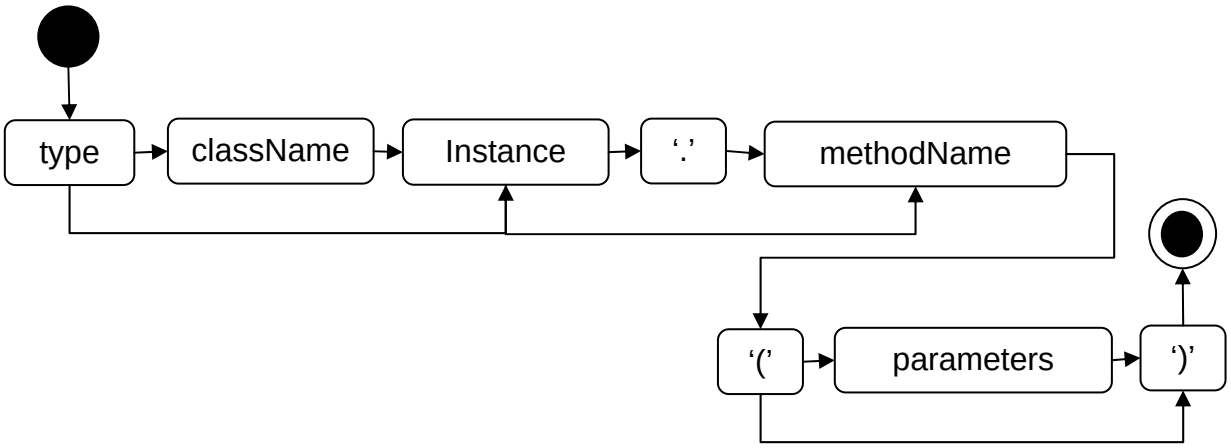

Figure 59: method syntax diagram

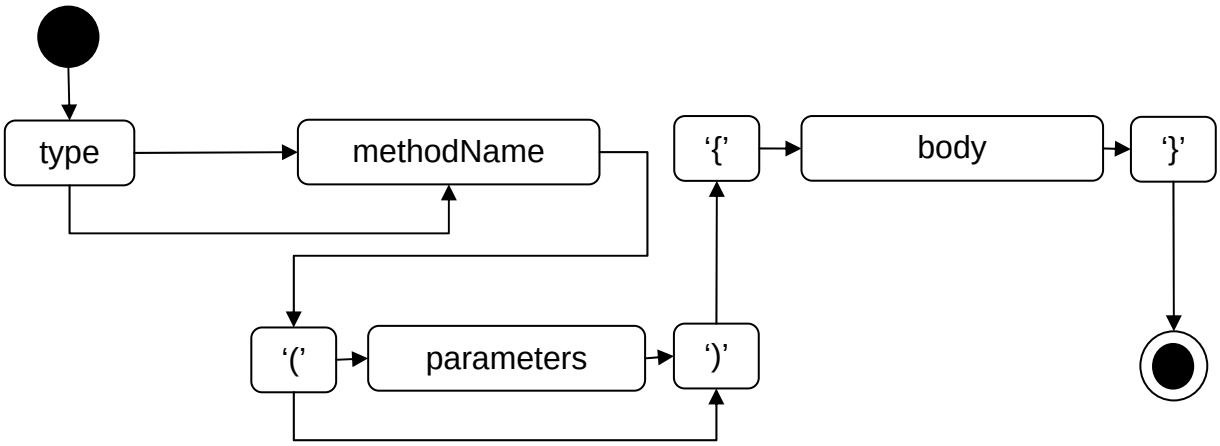

Figure 60: Method declaration syntax diagram

```
type      : Identifier;
method  : (type) (className)? (instance'.')?methodName ('('parameters?')'|'()')
                    ->^(Method ^(Type type) ^(Class className)? ^(Instance instance)?
                              ^(MethodName methodName) ^(Parameters parameters));
methodDeclaration : methodSignature'{'body'}'
                              -> ^(Method methodSignature ^(Body body));
methodSignature : type methodName'('parameters')'
                              -> ^(Type type) ^(MethodName methodName) ^(Parameters parameters)
                    | type methodName'()'
                              -> ^(Type type) ^(MethodName methodName);
valueComparison : className instance'.'attributeName'=='value
                              ->^(ValueComparison ^(Class className) ^(Instance instance)
                                        ^(Attribute attributeName) ^(Value value));
attribute : type className? (instance'.')?attributeName
                    -> ^(Attribute ^(Type type) ^(Class className)?
                              ^(Instance instance)? ^(Name attributeName));
parameters : parameter(','! parameter)*;
parameter : type? parameterName
                    -> ^(Type type)? ^(Name parameterName);
attributeName : Identifier;
attributeAlias : Identifier;
attributeSource : Identifier;
value : Identifier;
className : Identifier;
instance : Identifier;
methodName : Identifier;
body : (Identifier)*;
parameterName : Identifier;
Context :;
Name :;
Body :;
ContextRules :;
ContextRule :;
Source :;
Involves :;
InvolvedClasses :;
ContextState :;
Precondition :;
StateTransitionRule :;
ContextOccurrence :;
ContextOutParameters :;
Predicate :;
PredicateName :;
PredicateParameters :;
AttributeAlias :;
MethodAlias :;
Alias :;
ValueComparison :;
MethodSignature :;
Attribute :;
Value :;
Class :;
Instance :;
Method :;
MethodName :;
Parameters :;
Parameter :;
ParameterName :;
Type :;
```


Figure 61: JCOOL common syntactical rules

6.1.1 Context

Context is the JCOOL language construct that makes possible to define container for logically related contextual information. Its semantic is therefore very similar to the homonym concept defined in the 3rd chapter and consists of a vocabulary of those structural and behavioral features (attributes and methods) which contributes to the definition of a context. In the JCOOL language a context provides aliases for those target system attributes and methods it refers to in order to reduce the coupling between the base application and the context monitoring activities. Its syntactical structure is given by the grammar rules depicted in Figure 63 where the *contextDeclaration* rule is the starting rule for the Context construct. It states a Context consists of a *contextPresentation*, characterized by a context name, followed by the references of the classes involved by the context (*involvedClasses*), and a context body (*contextBody*).

The context body consists of sequence of context rules wrapped by curly brackets where each rule is separated with a ';' character from the others. A context rule consists of an alias for an existing attribute or an existing method of the involved classes. These aliases are respectively represented by the *attributeAliasing* and *methodAliasing* rules. An attribute alias can either directly refer to an existing attribute, by means of the *attribute* rule, or can be defined by the means of a function through the *methodDeclaration* rule. A method alias instead always refers to an existing method of the involved classes throughout the *method* rule.

Figure 62 depicts an example of Context definition. The proposed example consists of a Context named "SimpleContext" which involves the ClassA and ClassB classes.

```
Context SimpleContext involves ClassA, ClassB {
      attribute: ClassA *.attribute;
      attributeB: int sourceMethod(){
                        return attribute + 2;
                  }
      int method(int p1, String p2):
            int ClassA *.simpleMethod(int par1, String par2);
}
```

Figure 62: JCOOL Context example

The body of the context contains all those attributes and methods of the involved classes which are of interest for the definition of the given context. These are expressed in terms of an alias followed by the ":" character and the related attribute or method signature. The "*" character can be used as jolly character in order to express any character sequence. The alias can also be followed by round brackets containing a list of variables which can be used to expose out the monitors which uses the context a subset of the parameters used in the signature of the related method.

The context defined in Figure 62 consists of two attribute aliases, *attribute* and *attributeB*. The former refers to the *ClassA.attribute* structural feature while the latter is defined by means of a method returning a value obtained by an operation over the previously defined attribute. The context finally defines an alias for the *ClassA.simpleMethod* that exposes out all its parameters thanks to the *p1* and *p2* variables.

Figure 64 depicts the abstract syntax tree representation of the context depicted in Figure 62 that has been obtained with the ANTLR parser for JCOOL.

```
contextDeclaration    : contextPresentation^ involvedClasses contextBody;
contextPresentation   : CONTEXT contextName
                            -> ^(Context ^(Name contextName));
contextName           : Identifier;
involvedClasses       : INVOLVES involvedClass
                            -> ^(Involves involvedClass);
involvedClass         : Identifier (','! involvedClass)*;
contextBody           : '{'contextRules'}'
                            -> ^(ContextRules contextRules);
contextRules          : ((contextRule)';'!)*;
contextRule           : attributeAliasing
                            -> ^(AttributeAlias attributeAliasing)
                      | methodAliasing
                            -> ^(MethodAlias methodAliasing);
attributeAliasing     : attributeAlias':'(attribute)
                            -> ^(Alias attributeAlias) ^(Source attribute)
                      | attributeAlias':'(methodDeclaration)
                            -> ^(Alias attributeAlias) ^(Source methodDeclaration);
methodAliasing        : methodAlias':'method
                            -> ^(Alias methodAlias) ^(Source method);
methodAlias           : methodSignature;
```

Figure 63: JCOOL Context syntactical rules

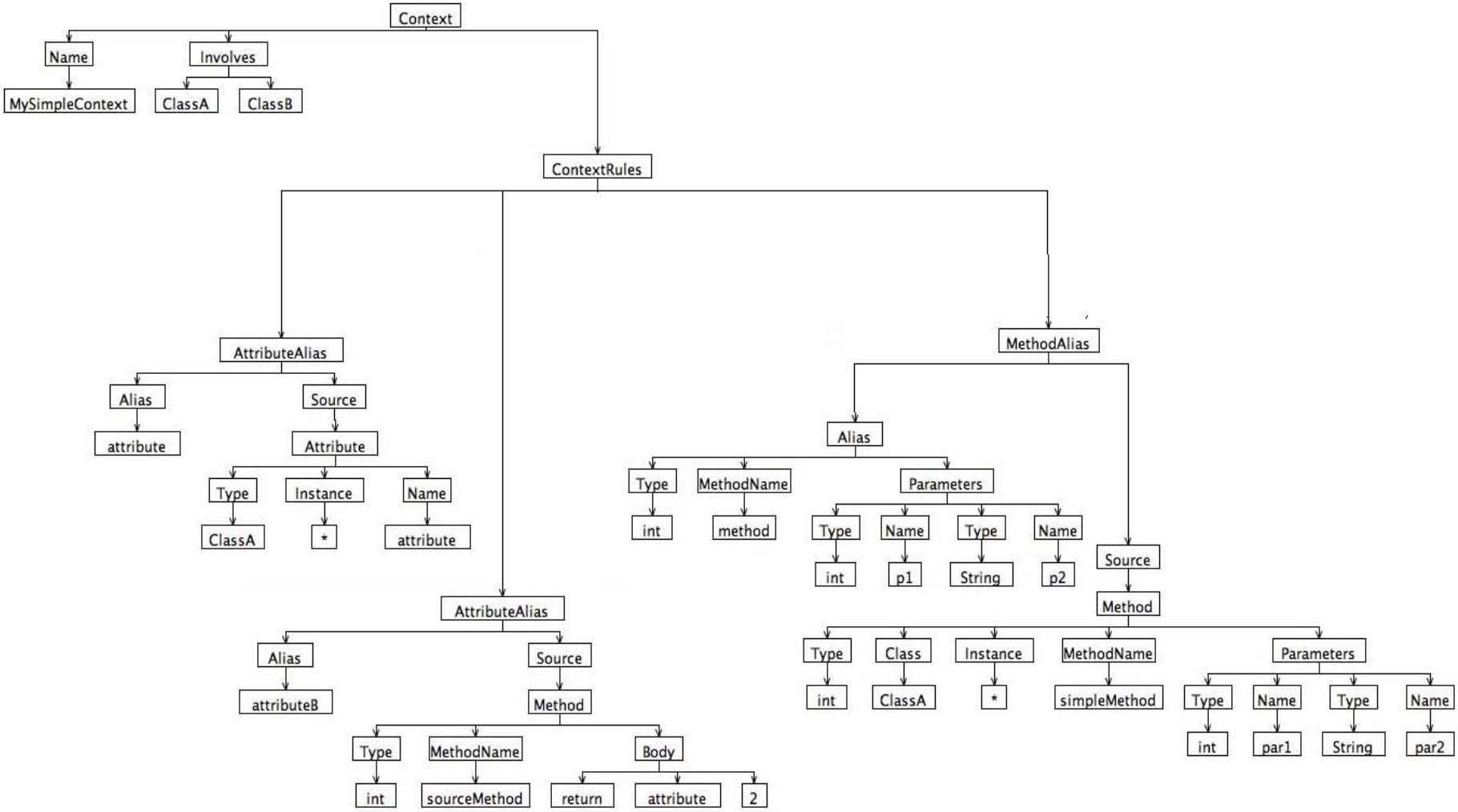

Figure 64: an example of Abstract Syntax Tree for JCOOL Context

6.1.2 Monitor

As its conceptual counterpart, represented by the homonym concept described in the 3rd chapter, the *Monitor* construct acts as container for adaptation triggers constrained by the information provided by contexts, or other monitors. These constraints are defined exploiting those aliases defined in the observed contexts so that the code implementing the monitoring concern is not coupled to the target system and can be easily reused into different systems.

The syntactical structure of the *Monitor* construct is given by the grammar rules depicted in Figure 67 where the *monitorDeclaration* rule represents the starting rule for this grammar. It states a monitor declaration consists of a *monitorPresentation*, characterized by the monitor name, followed by the references of the contexts and monitors observed by the monitor (*observedContexts*), and a body (*monitorBody*).

The body of a monitor consists of a sequence of rules (*monitorRules*) wrapped by curly brackets where each rule is separated by a ";" character from the others. Each rule consists of two parts: a context state (*contextState*), representing a state of interest with respect to the information provided by the observed contexts, and a state transition rule (*stateTransitionRule*), representing the conditions that have to be verified by the observed contexts in order to consider related context state activated.

A context state is defined by a name possibly followed by a list of exposed parameters in round brackets that can be exploited in order to capture the whole or a subset of the context information that has caused the context-state activation. The ':-' characters separate a context state by its state transition rule which is the rule that, when verified, trigger the activation of the context-state.

```
Monitor SimpleMonitor observes SimpleContext {
      stateA:- (SimpleContext *.attribute ==1),
                [method(int par1, String par2){2}];
      stateB(i):- (SimpleContext i.attribute == 2);
}
```

Figure 65: JCOOL Monitor example

<ContextState>(<parameters>) :- <StateTransitionRule>

Figure 66: Context State Transition Rule example

A default-state with no transition rule is implicitly always defined.
A monitor is in a given state until the related transition rule holds while it is in the default state if none of its transition rules is verified. A state transition rule consists of a set of one or more predicates over the observed contexts and possibly other monitors combined with the logical operators ',' (AND), '|' (OR) and '!' (NOT). Each atomic predicate may either consinsts of a method invocation (identified by the *method* rule) or a value comparison among structural features (which is identified by the *valueComparison* rule). On the transition between two states a monitor may trigger the execution of one or more adaptors through the invocation of one of its entry points.
As mentioned monitors have a compositional characteristic so that the context-states of a monitor may have state transition rules which depend on the context-states of other monitors it observes.
Sometimes it may be necessary to detect precise sequence of events in order to consider a context in certain state. This can be done by means of predicates that can be considered the code counterpart of the *EventConstraint* concept defined in the 3rd chapter. To specify such a constraint in JCOOL square brackets must be used to enclose those predicates which represent events that must occur in the exact sequence they are written in. Operators ?, + and * can be used, like in regular expression, to express that a constraint should occur respectively: never or one time; at least one time; never or any time. Curly brackets can finally be used to enclose the exact number of times a constraint has to occur.
Figure 65 depicts an example of monitor definition. It consists of a name, which has to be unique among the monitors belonging to the same package, followed by the list of the *observed* contexts and monitors. The proposed example consists of a monitor named "SimpleMonitor" which involves the context named *SImpleContext*.
The SimpleMonitor monitor defines two context states respectively named *stateA* and *stateB*. The former is activated as soon as the value related to the structural feature named *attribute* of the observed *SimpleContext* is equal to one and the behavioural feature named *method* has been invoked twice. The latter context-state holds as soon as the value related to the structural feature named *attribute* of the observed *SimpleContext* is equal to two. However differently by the first context-state the *stateB* context-state also captures a picture of the current context information that has caused

its activation by means of the free variable named *i* which takes as value a copy of the *SimpleContext* as soon as the *stateB* is activated.

Figure 68 depicts the abstract syntax tree representation of the described monitor that has been obtained with the ANTLR parser for JCOOL.

```
monitorDeclaration      : monitorPresentation^ observedContexts monitorBody;
monitorPresentation     : MONITOR! monitorName^;
monitorName             : Identifier;
observedContexts        : OBSERVES^ observedContext;
observedContext         : Identifier (','! observedContext)*;
monitorBody             : '{'monitorRules'}'
                                -> monitorRules ;
monitorRules            : ((monitorRule))*;
monitorRule             : contextState':-'stateTransitionRule*';'
                                -> ^(StateTransitionRule ^(ContextState contextState)
                                 ^(Precondition stateTransitionRule))
                        ;
contextState            : contextOccurrence(OPEN parameters CLOSE)?
                        -> ^(ContextOccurrence contextOccurrence)
                                 ^(ContextOutParameters parameters)?
                        ;
contextOccurrence       : Identifier;
contextOutParameters    : (contextOutParameter(','contextOutParameters)*);
contextOutParameter     : type value
                                ->^(Parameter ^(Type type) ^(ParameterName value));
stateTransitionRule     : predicate ;
atomicEvent             : sequence|method|valueComparison;
valueComparison         : className instance'.'attributeName'=='value
                                ->^(ValueComparison ^(Class className) ^(Instance instance)
                                ^(Attribute attributeName) ^(Value value));
predicate               : innerPredicate ((AND||OR||NOT)innerPredicate)*;
innerPredicate          : OPEN predicate CLOSE -> ^(Predicate predicate)
                        | atomicEvent ;
sequence                : '['sequencePredicate']'
                        -> ^(SequencePredicate sequencePredicate);
sequencePredicate       : innerSequencePredicate ((AND||OR||NOT)innerSequencePredicate)*;
innerSequencePredicate : OPEN predicate CLOSE '?'
                                 -> ^(Predicate predicate NEVER_OR_ONE_TIME )
                        | OPEN predicate CLOSE '+'
                                -> ^(Predicate predicate AT_LEAST_ONE_TIME)
                        | OPEN predicate CLOSE '*'
                                 -> ^(Predicate predicate ANY_TIME)
                        | OPEN predicate CLOSE occurrences
                                ->^(Predicate predicate occurrences)
                        | innerPredicate
                        | atomicEvent ;
occurrences             : '{'occurrence'}'
                                -> ^(Occurrences occurrence);
```


Figure 67: JCOOL Monitor syntactical rules

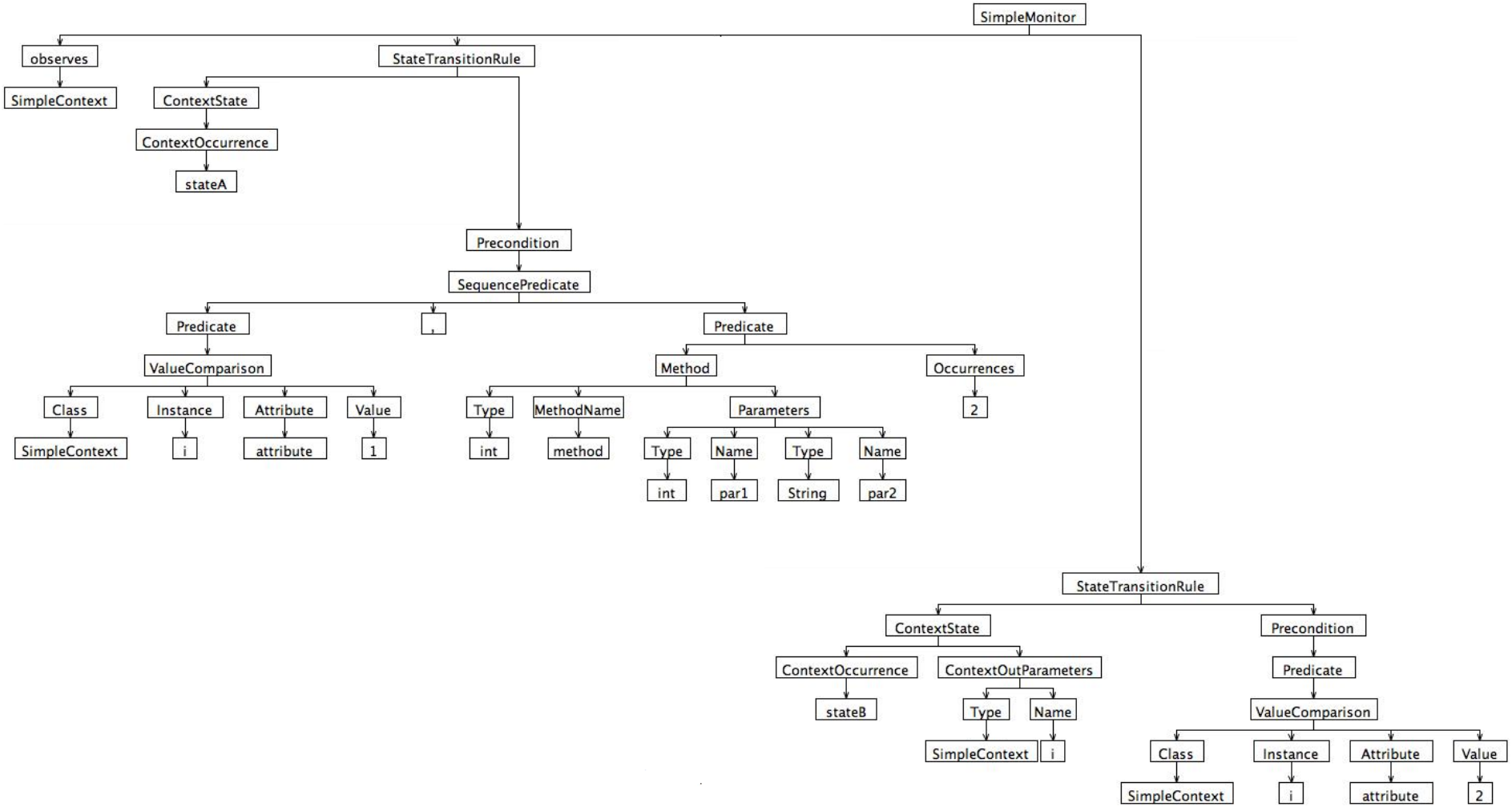

Figure 68: an Example of Abstract Syntax Tree for JCOOL Monitor

6.1.3 Adapter and AdaptationLayers

As its conceptual counterpart, represented by the homonym concept defined in the 3rd chapter, the *Adapter* construct acts as a container for logically related context adaptation mechanisms. The syntactical structure of the *Adapter* construct is given by the grammar rules depicted in Figure 69 where the *adapterDeclaration* rule represents the starting rule of this grammar. It states an adapter consists of an *adapterPresentation*, characterized by an adapter name, followed by the references of the monitors whose context-states may trigger the adapter (*driverMonitors*), and a body (*adapterBody*). Each adapter is identified by a unique name and may be driven by one or more monitors as well each monitor may drive several adapters. On the transition between two context-states a monitor may trigger the execution of one or more adapters through the invocation of its entry points.

The body of an adapter consists of a sequence of rules (*adapterRules*) each one representing a trigger that can be activated by the occurrence of a context-state belonging to the *driverMonitors*. Each rule thus consists of two parts: the former represents the monitor context-state which triggers the activation of the rule while the latter, called *triggerBody* and wrapped by curly brackets is itself composed by three parts: *adaptationMechanism, incomingAdaptation*, and *outgoingAdaptation*. Parameters can be passed to the adaptation action after a transition rule is evaluated and fired and the related context-state activated. These parameters are free variables which take the values of those objects which have verified the state transition rule.

The *adaptationMechanism* block wraps in curly brackets the code which manages the behavioural variations that have to be activated in response of the context-state occurrence. The incoming adaptation block wraps in curly brackets the Java code that has to be executed, maybe exploiting the introduced behavioural variations, as soon as the related context-state occurs while the outgoing adaptation block the Java code that has to be executed, maybe exploiting the introduced behavioural variations, at the related context-state outgoing event.

Figure 70 depicts an example of JCOOL Adapter named *SimpleAdapter* which is driven by the SimpleMonitor defined in Figure 65. The *SimpleAdapter* consists of two adapter ruels, *SimpleMonitor.stateA* and *SimpleMonitor.stateB*, which respectively refer to the *stateA* and *stateB* context-states of the Monitor named *SimpleMonitor* which drives it. Both the adapter rules have no incoming/outgoing adaptations and consists in the

activation of an adaptation mechanism represented by the behavioural variation named *SimpleLayer*.

Behavioral variations, implemented by the *AdaptationLayer* construct, consist of a set of structural or behavioural inserts and bindings that may affect a class or a particular class instance passed as parameters to the layer through the related adapter. To this end, when defining an adaptation layer it is necessary to indicate the class involved by the layer. Each behavioural variation has to start with the class type to which it is supposed to be applied. Adaptation layers provide two methods: *isApplicable* and *apply* the adapter can exploit respectively to state if a layer is applicable on a certain class instance and, conseguently, to apply the layer on it or not. When defining a layer the former method has to be properly overridden in order to implement the desired behaviour. The adapter can apply a layer on a specific object, instances of the Class involved by the layer, by passing it as parameter of the *apply* method. When the appy method is invoked with no parameters the layer is activated on all the instances of the involved class.
An adaptation layer is active until the triggering monitor remains in the related context-state. When an adaptation layer is no longer active the behavioural variations it has introduced are automatically removed. As a consequence the introduced binding returns to their original value while behaviorural and structural inserts are automatically removed.
Figure 71 depicts an example of adaptation layer named *SimpleLayer* involving the *ClassA*. When applied it introduce an alternative implementation of the *ClassA* method named *simpleMethod* which substitutes the previous one. As mentioned adaptation layers may affects specific instances of a given class so that, in a given time, different objects of the same class may have different implementations of the same methods depending on their actual context. It is the adapter applicating the layer which has to chose how to do it. For example in both the entry points of the *SimpleAdapter* depicted in Figure 70 the *SImpleLayer* is instantiated and applied. However while the application of this layer in the *SImpleMonitor.stateA* entry point is related to all the existing instances of the classes it involves, in the *SimpleMonitor.stateB* entry point the layer is activated only for the instance passed by the monitor to the adapter through the variable *i*. Figure 72 depicts the abstract syntax tree representation of the described adapter that has been obtained with the ANTLR parser for JCOOL.

```
adapterDeclaration    : adapterPresentation^ driverMonitors adapterBody;
adapterPresentation   : ADAPTER! adapterName^;
adapterName           : Identifier;
driverMonitors        : DRIVENBY driverMonitor
                            -> ^(DrivenBy driverMonitor);
driverMonitor         : Identifier (','! driverMonitor)*;
adapterBody           : '{'adapterRules'}'
                            -> adapterRules;
adapterRules          : ((adapterRule))*;
adapterRule           : monitorName'.'contextState'{'triggerBody'}'
                            -> ^(MonitorTrigger ^(MonitorName monitorName)
                            contextState triggerBody);
triggerBody           : adaptationMechanism incomingAdaptation outgoingAdaptation;
incomingAdaptation    : 'in:"{'body?'}'
                            -> ^(IncomingActivities body)?;
outgoingAdaptation    : 'out:"{'body?'}'
                            -> ^(OutgoingActivities body)?;
adaptationMechanism   : body?
                            -> ^(AdaptationMechanism body)?;
```

Figure 69: JCOOL Adapter Syntactical rules

```
Adapter SimpleAdapter drivenBy SimpleMonitor applies SimpleLayer {

      SimpleMonitor.stateA {

            //layer applications
            AdaptationLayer a = new SimpleLayer();
            if(a.isApplicable())
                  a.apply();

            //Incoming Adaptation
            in:{ }

            //Outgoing Adaptation
            out:{ }

      }

      SimpleMonitor.stateB(i) {

            //layer applications
            AdaptationLayer a = new SimpleLayer();
            if(a.isApplicable(i))
                  a.apply(i);
            //Incoming Adaptation
            in:{ }

            //Outgoing Adaptation
            out:{ }

      }
}
```

Figure 70: JCOOL Adapter example

```
AdaptationLayer SimpleLayer involves ClassA {
            //isApplicable method overriding
            public boolean isApplicable(ClassA instance){
                  //to be overridden
            }
            //Inserts and Bindings...
            public void ClassA *.simpleMethod(int par1, String par2){
                  //Alternative method implementation
            }
}
```

Figure 71: JCOOL AdaptationLayer

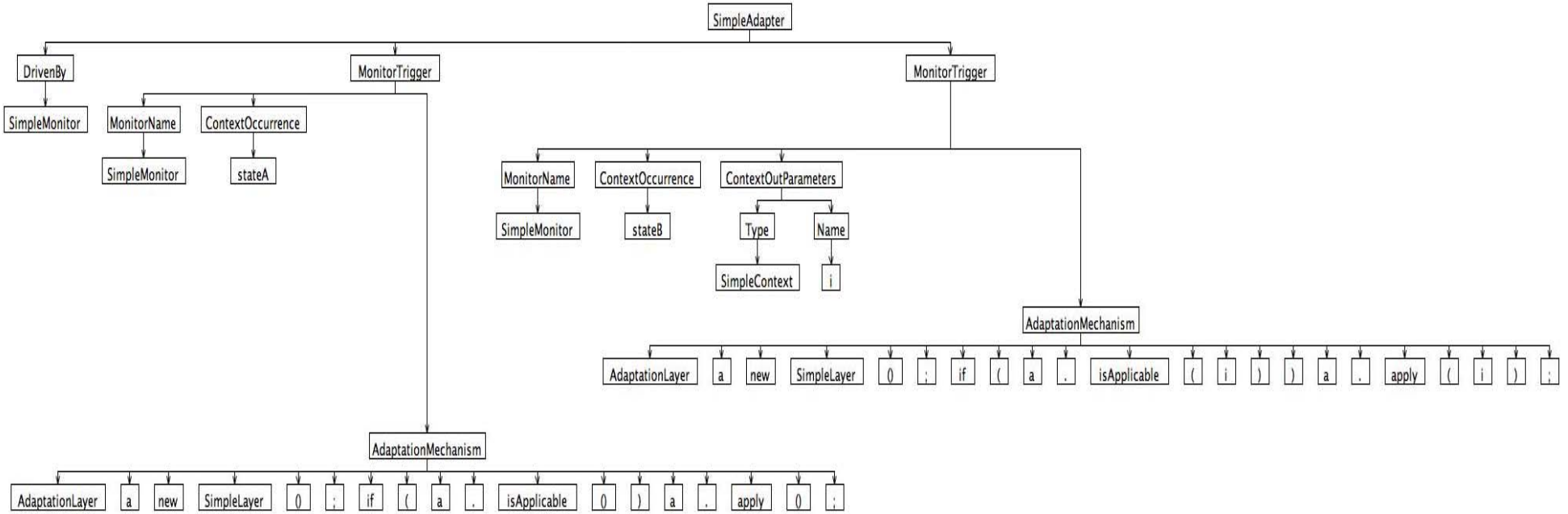

Figure 72: an Example of Abstract Syntax Tree for JCOOL Adapter

7 Case Study: The Traveler Service Application

In this chapter we propose a simple application scenario aimed at providing a concrete example of how our modeling and development environment can be exploited to introduce context awareness characteristics into a, possibly already existing, and independently designed target application. The chosen scenario relates to the traveler service application we have already introduced in [3]. In the first section of this chapter we depict how the Eclipse UML plugin can be exploited to produce a proper UML model of the base application. Then, in the second section, we depict how these models can be referenced by our modelling framework to produce the models of the desired context aware features. The obtained models can then be used as input of model transformation workflows aimed at producing code and other desired artefacts (i.e. documentation, metrics, etc.).

7.1 The Traveler Service Application Base Models

Consider a generic Traveler Service application that travelers within an airport can access through a wireless connection using their own portable devices. The application displays a GUI through which travelers may use basic services such as: ticket purchase, automatic check-in, finding a restaurant in the airport, etc.

Figure 73 depicts the static structure of this application. The *AeroGui* Class represents the component responsible to realize the man machine interface of the traveler service. For the sake of simplicity in this example we only depict three methods which are the handle the user invoke to exploit the services of automatic check in, and restaurant finding. The *UserInformation* class represents the actual user of the traveler service application. In this simple example we are only interested in two methods of this class: the former named *getCompleteName* returns a *String* parameter representing the name of the actual user while the latter, the *getOwnedTicket* returns a *Ticket* instance which represents the ticket already bought by the actual user, if it exists. For the sake of simplicity we assume the user may have at most one ticket.

The *CheckinManager* class is responsible to handle the check-in operation while the *RestaurantFinder* class represents the local front-end of a possible already-existing web-service the *AeroGui* class exploits to look for the available local restaurants.

The manager of an airport might want to customize this application by adding some context-aware enhancement to its basic services.

For example, the restaurant finding service could be enhanced by automatically filling in the *cuisinePreferences* parameter, using for this purpose information provided by an existing *user profiling* service. Moreover, the returned list of restaurants could be arranged according to how much time the traveler may presumably spend at the restaurant (which could depend on whether s/he has successfully checked-in, and how much time remains for the scheduled flight departure): if the traveler has few time to spend, fast food restaurants could be displayed first, or even a warning could be given if the boarding time is approaching.

In the following paragraphs we depict how these context-aware characteristics can be modeled independently of the models of the base system exploiting our eclipse based context-modeling framework.

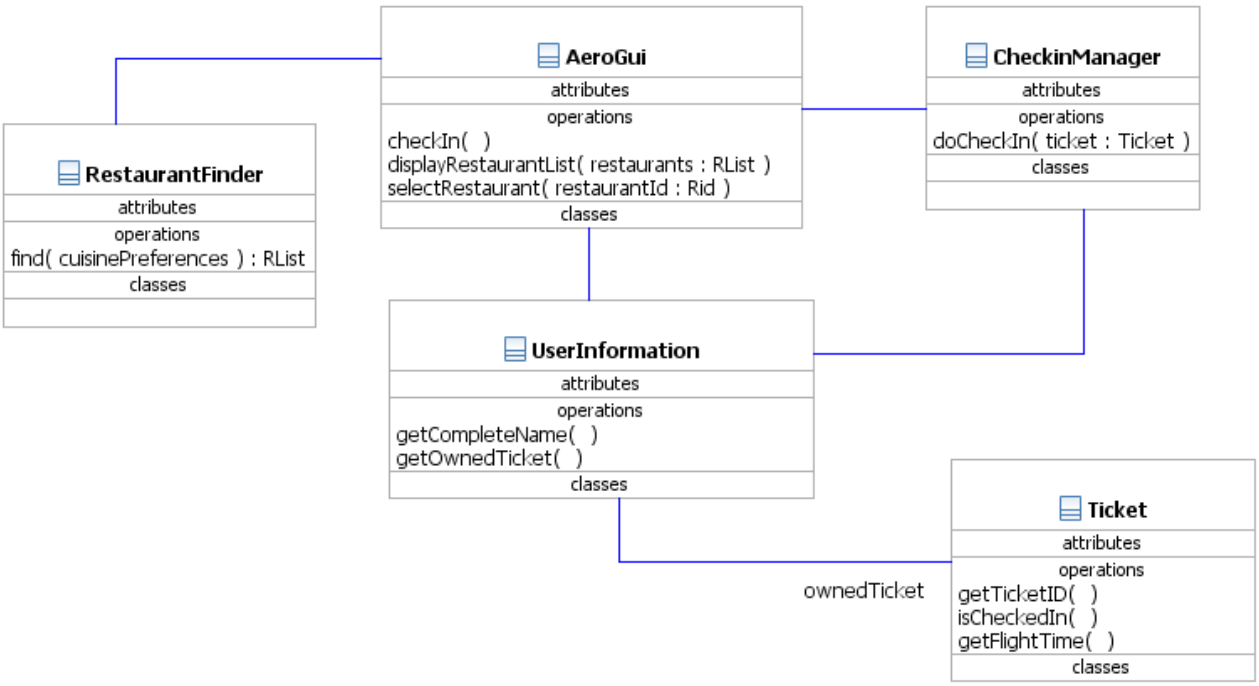

Figure 73:Travelers Service Application Class Diagram

7.2 The Context Models

Starting from the UML class diagram of this simple example we can use the CAMEL editor to model the new aforementioned context aware behaviours. Figure 74 depicts the first step of the context modeling workflow in the CAMEL editor. In this figure an Eclipse instance has been executed having a workspace containing an *applicationModels* folder

with the uml class diagram depicted in Figure 73 and a *contextModels* folder containing a new empty context model named *travvelerContextModel.contextmodel*.

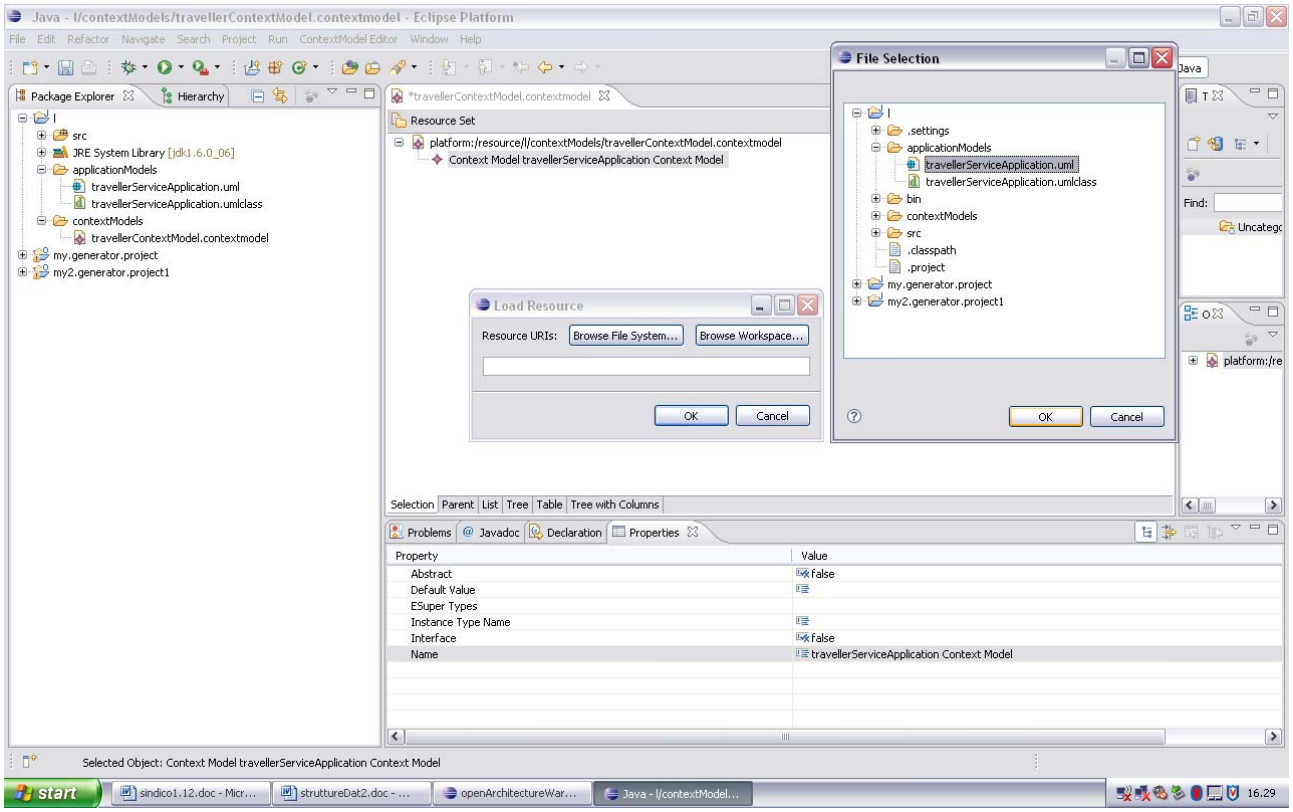

Figure 74: creating the context model

The figure also depicts how the user can load in the CAMEL editor an external UML model, in this case the traveler service class diagram, residing in the same workspace. This fundamental step makes possible to associate the entities of the context model to the entities of the loaded uml model. In this example we are actually interested in two different kinds of context informations: those related to the current time, and those related to the user. Both these two entities can be modelled with *StateBasedContexts*. To this end two state based contexts have been created named *Time* and *UserProfile*. The former is depicted in Figure 75 and consists of a context attribute named *currentTime* which has as source the *currentTime* attribute of the *Time* class. Because the source of a context attribute is modeled as a structural feature of the *ContextAttribute* EClass its value can be seen and edited in the properties tab in the bottom of the CAMEL editor (Figure 75). The *UserProfile* state based context is depicted in Figure 76. It consists of a context attribute named *user* which has as source the actual value of the reference between the *AeroGui* and the *UserInformation* instances. Moreover, because the context information concerning the user are scattered among the system components (*Ticket*, etc.), operations are added to this context attribute

(*getCompleteName*, *getFlightTime*, *isCheckedIn*) each one encapsulating the logic related to obtain a specific context information. In order to hide these operations to the rest of the model new context attributes are then introduced (*completeName*, *flightIme*, *isCheckedIn*, etc.) that have as source the out parameter of the related user operation (Figure 77).

Figure 78 depicts the modeling of a composite context named *UserActivities* which consists of an event based and a state based context. The former consists of two context events named *locatingRestaurant* and *checkingIn* which respectively relate to the invocation of the *AeroGui::selectRestaurant(restaurantId: Rid)* and *AeroGui::checkin()* methods. The state based context instead (Figure 79) consists of a context attribute named *selectedRestaurant* which has as source the restaurantId parameter passed to an invoked *AeroGui::selectRestaurant(..)* method.

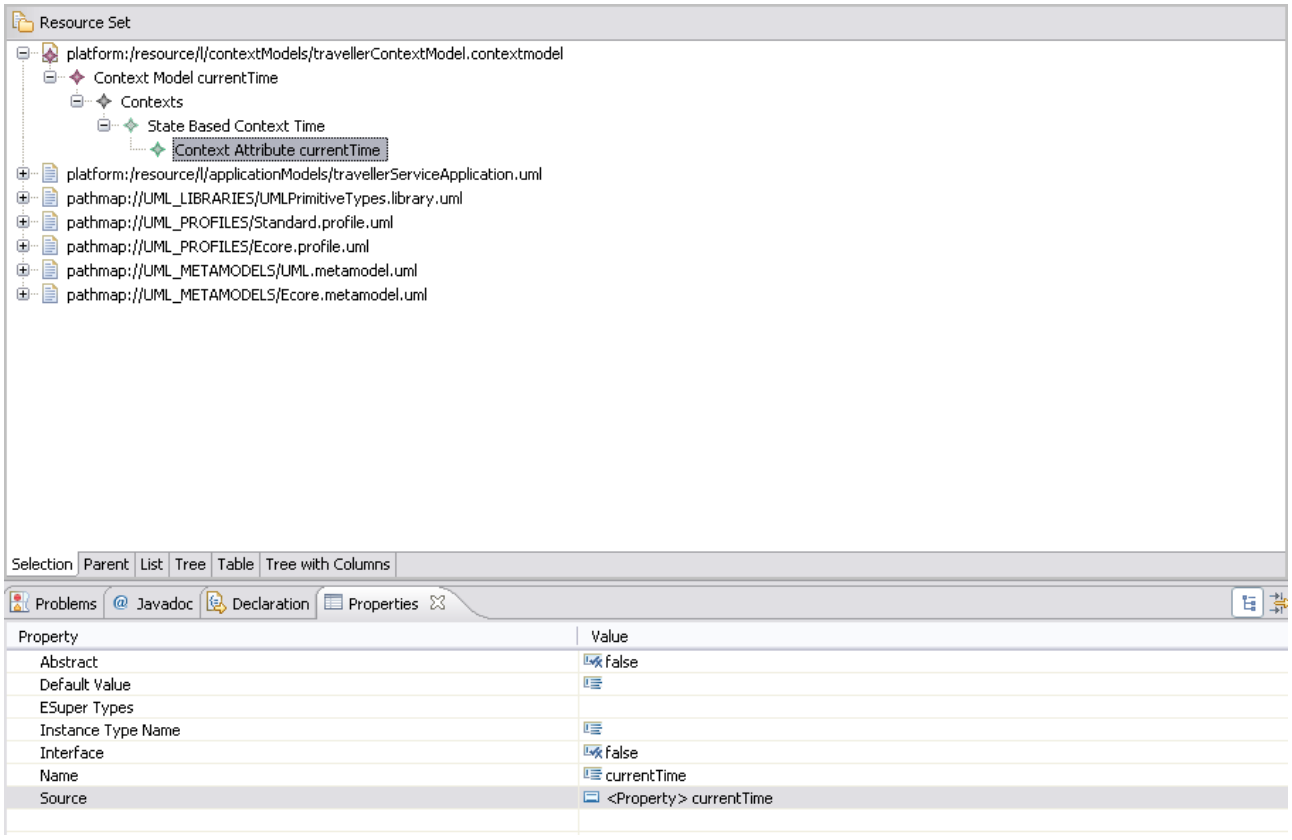

Figure 75: The Time Contextd

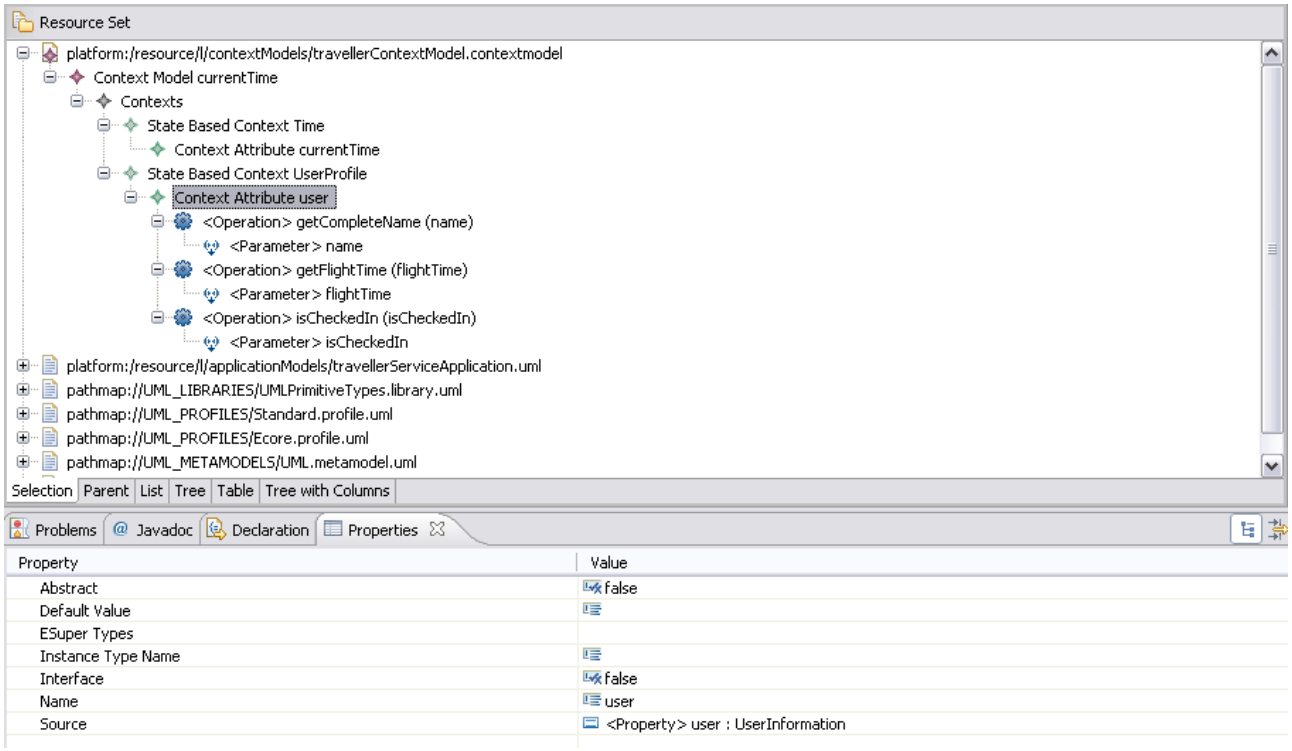

Figure 76: The UserProfile Context

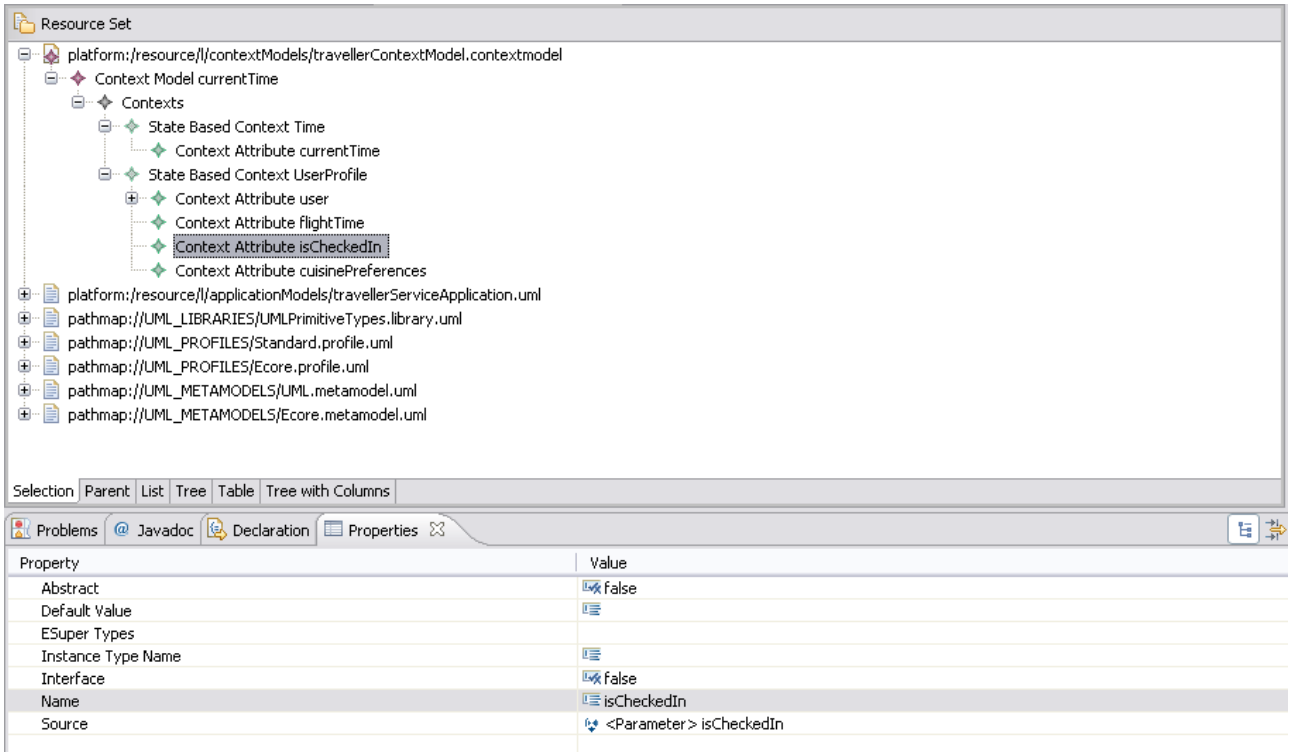

Figure 77: The User Profile Context

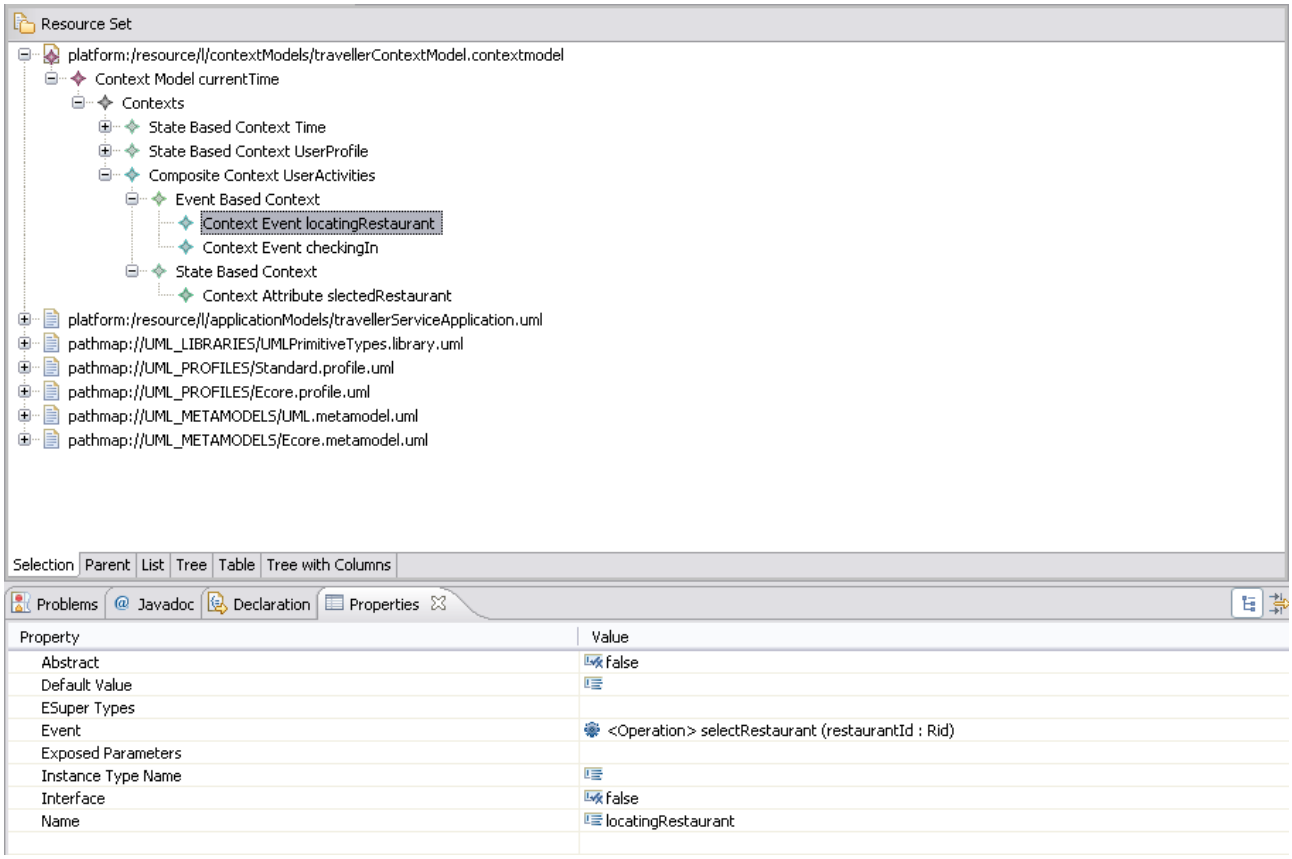

Figure 78: The UserActivities Context

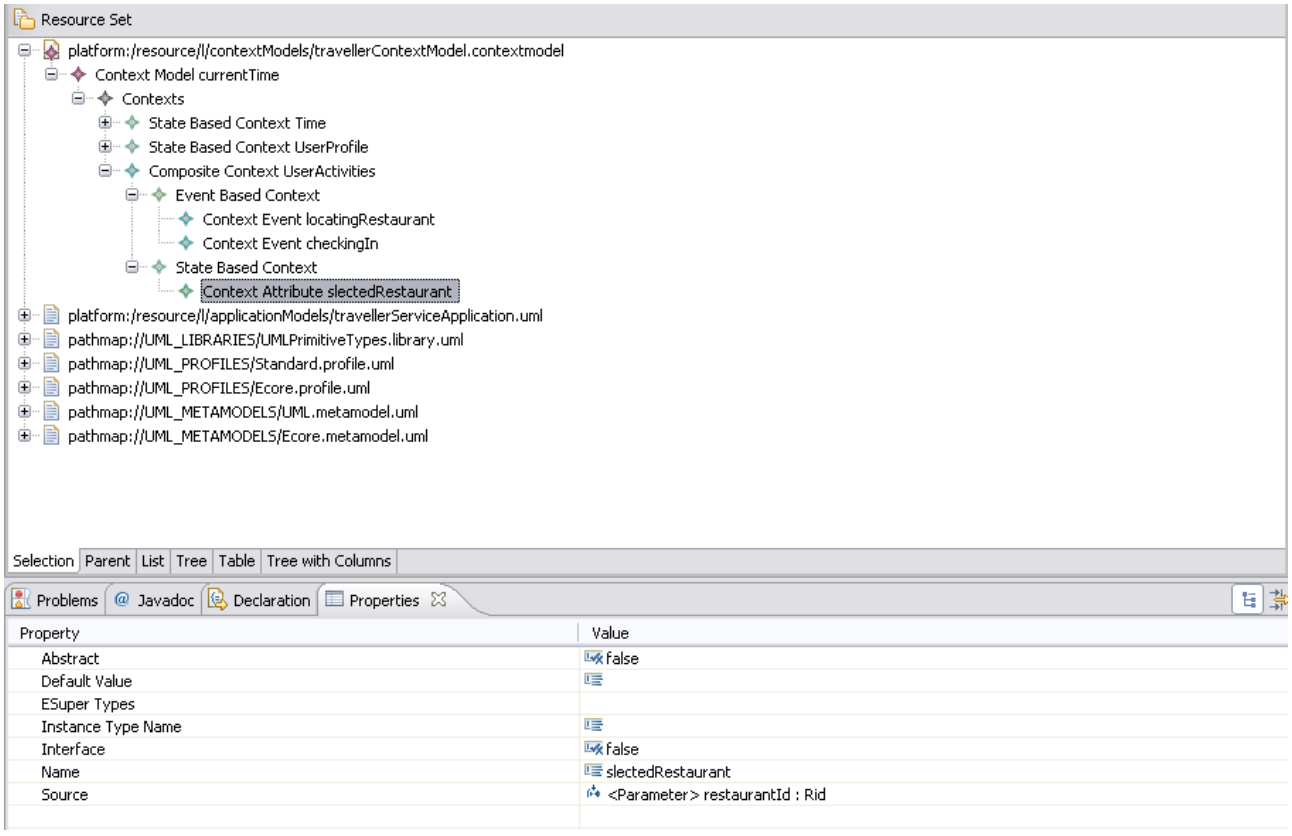

Figure 79: The User Activities Context

7.3 The Context Adaptation Triggering Models

Once contextual information has been modelled with state and event based contexts, we have to model those conditions over the context information that identifies interesting context states. We have thus to model those entities we have called context monitors.

In this example scenario we are interested in capturing the following context states:

- the user has just selected a restaruant link (he/she is looking for a restaurant);
- it actually remains few time before the user's flight;
- it is lunch time and there is more than an hour to the user's flight.

In Figure 85 we depict the model of a context monitor, named *UserMonitor*, which contains the models of these context states which are respectively named: *locateRestaurant*, *fewFreeTime* and *lunchtime*.

The model of the former context state is depicted in Figure 80 and Figure 81. It consists of an event constraint which captures the occurrences of the *locatingRestaurant* context event so that the *locateRestaurant* context state goes active whenever the user is actually looking for a restaurant.

The modeled context-state also exposes the reference to the *AereoGui* object whose *selectRestaurant* method has been actually invoked, which can be exploited by the Adapters driven by this context-state throught the alias *guiInstance* specified by the InstanceReference attribute (Figure 81).

The *fewFreeTime* context-state is a little more complex. It consists of two state based constraints combined together by an AND operator. The former constraint checks if the check-in operation has been successfully performed by the user, the latter checks if the flight will be in less than an hour. If these two conditions are verified the contex-state is activated (Figure 83).

The *lunchtime* context state finally consists of a set of context constraints which basically verify that: the user has successfully checked in; the value of the currentTime context attribute is between the 12:00 and 14:00 and that there is at least one ore to the flight (Figure 84).

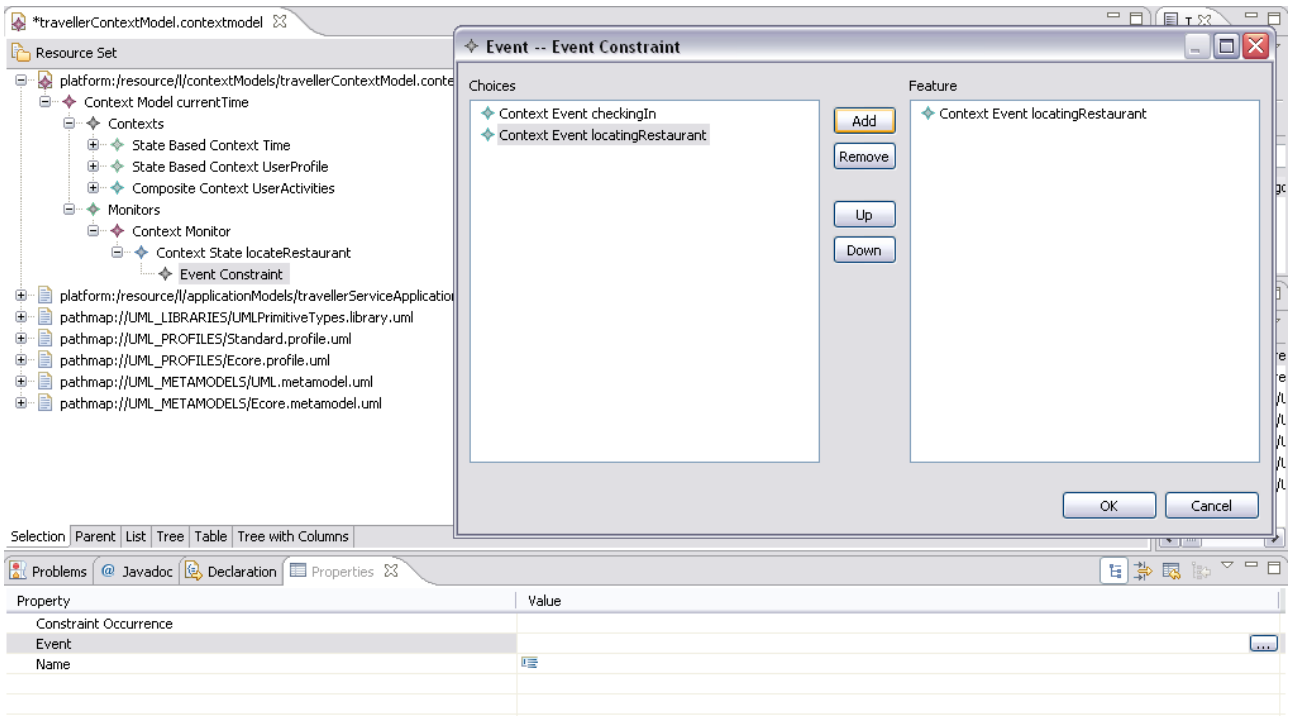

Figure 80: UserMonitor – locateRestaurant

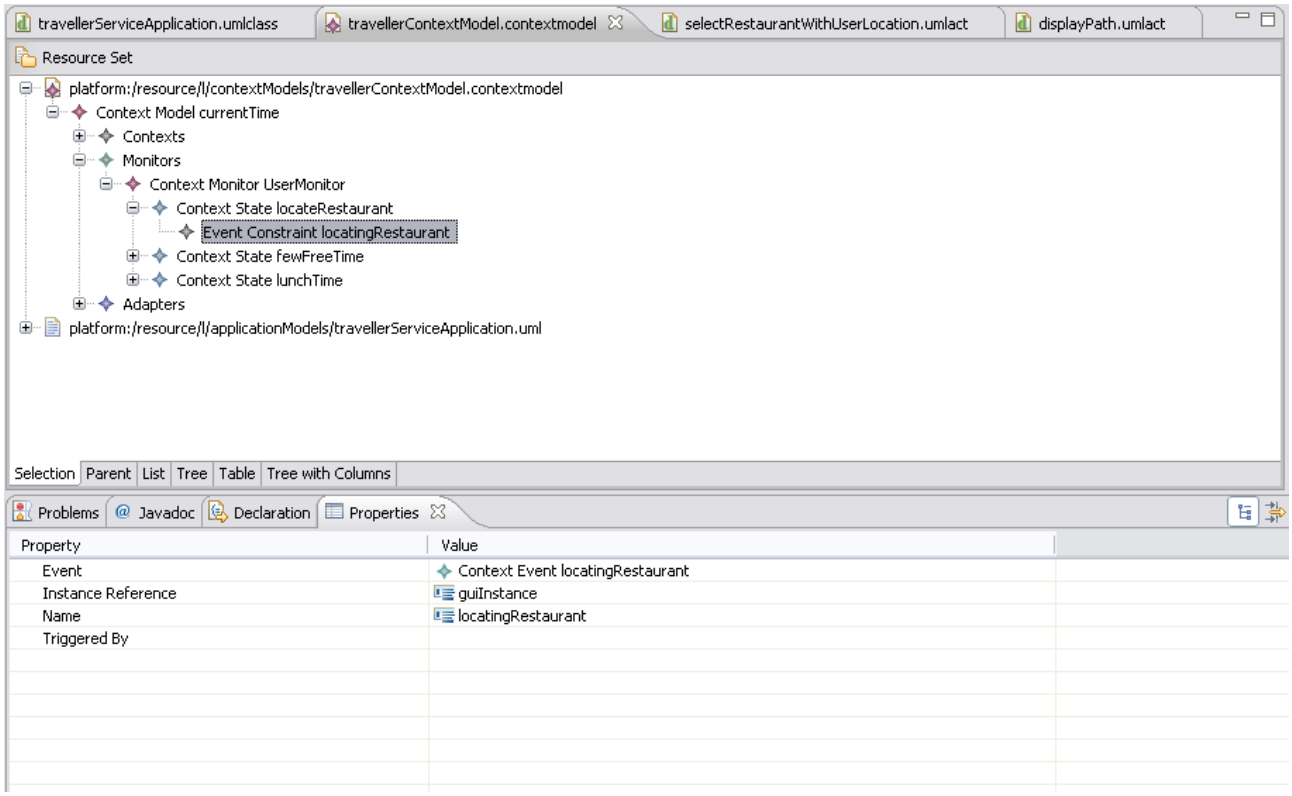

Figure 81: UserMonitor – locateRestaurant context state

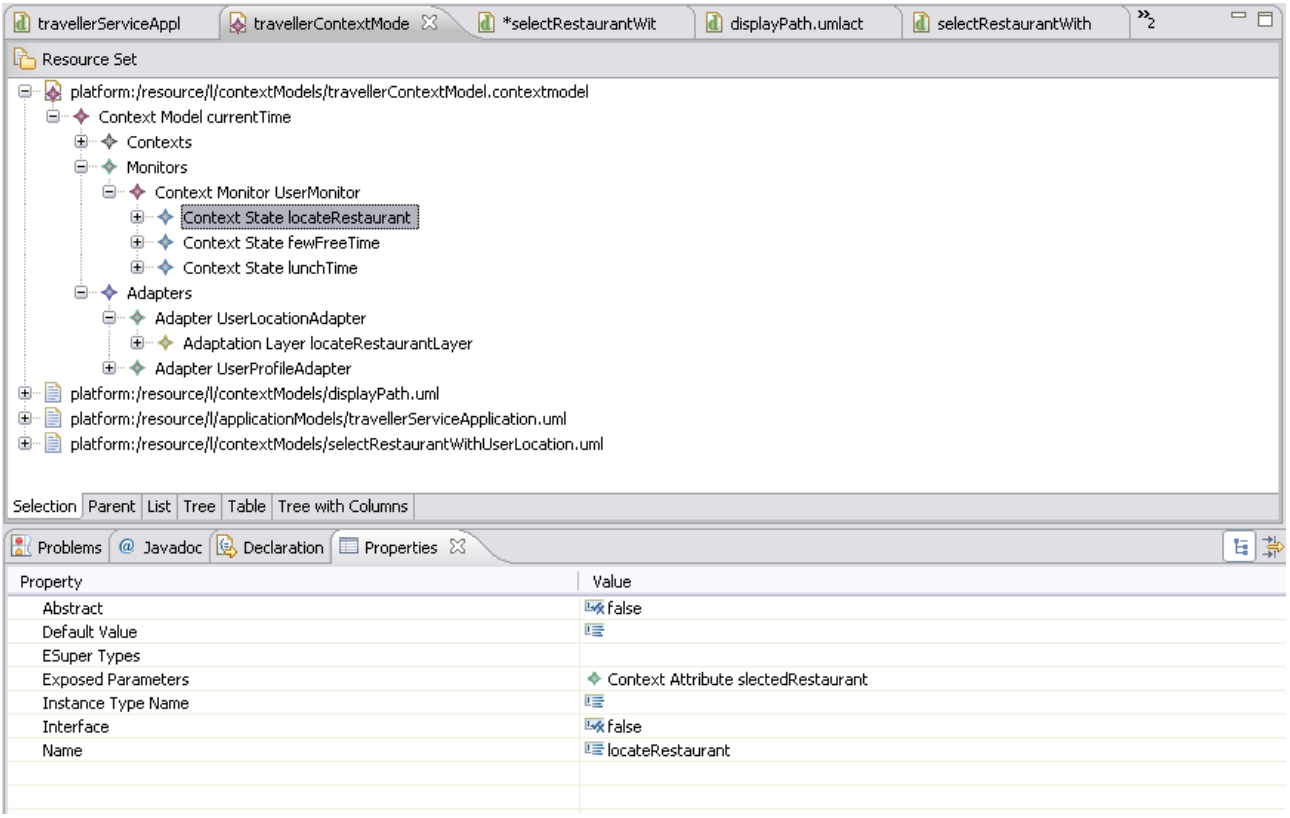

Figure 82: UserMonitor – locateRestaurant context state

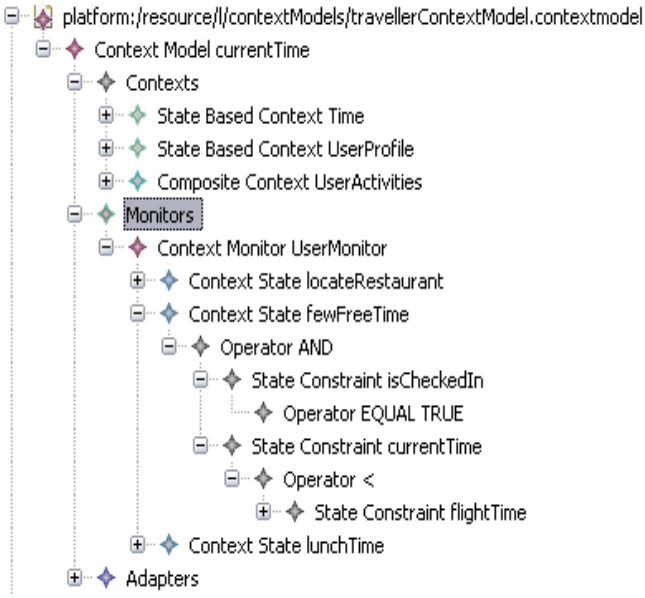

Figure 83: UserMonitor – fewFreeTime context state

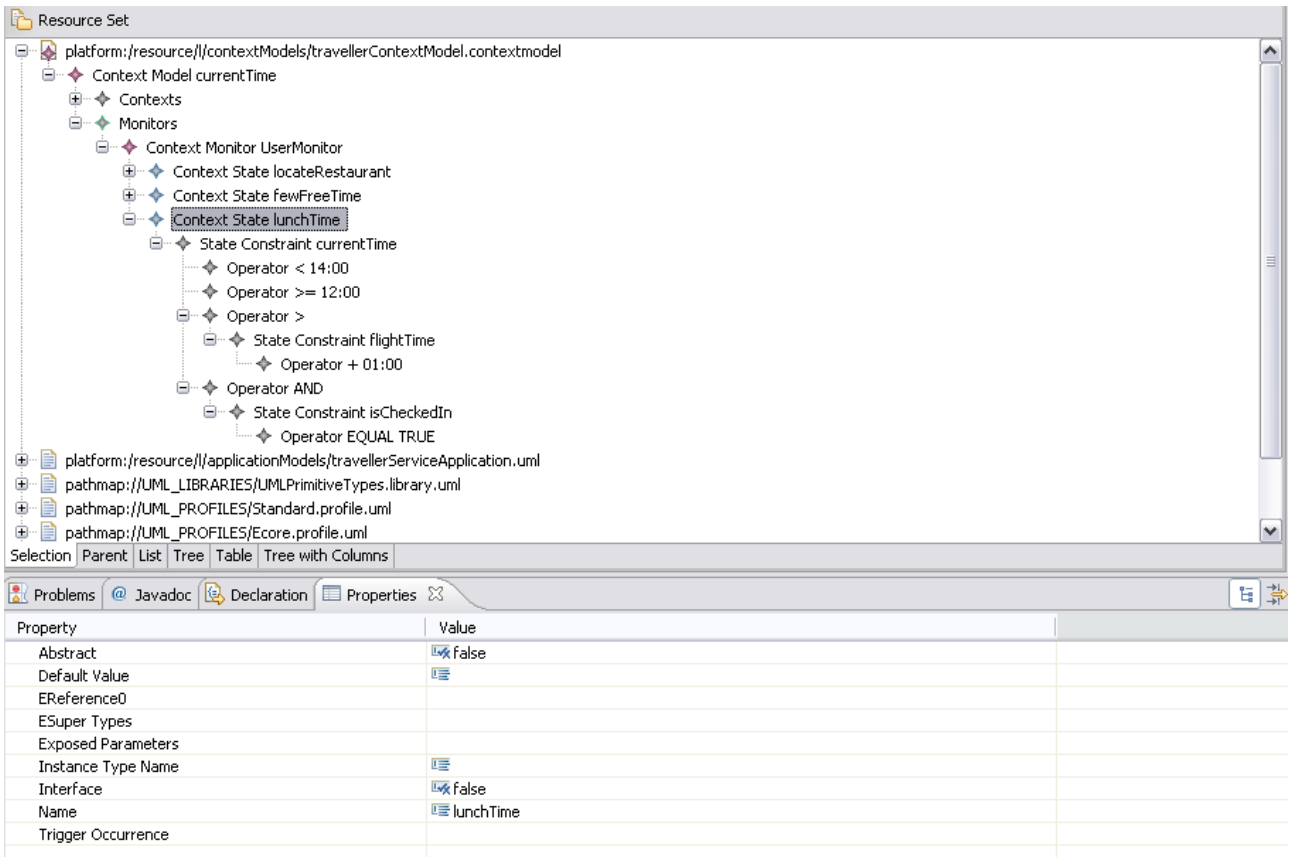

Figure 84: UserMonitor – lunchtime context state

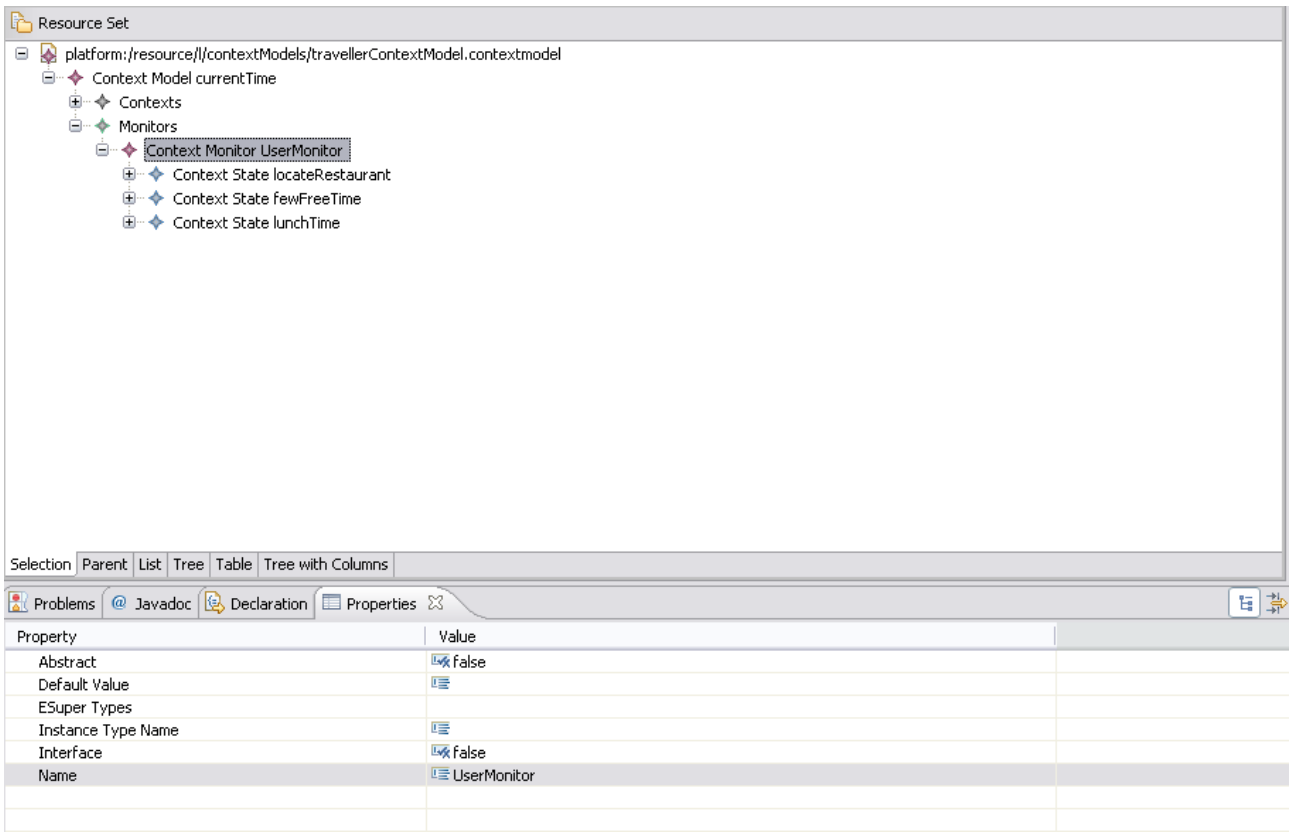

Figure 85: The UserMonitor Context Monitor

7.4 The Context Adaptation Models

As explained in section 3.2, context monitors observes contextual information checking for those conditions that identify interesting states of the system's context. In order to react to the context changes that are captured by monitors by the means of context-states, adapter can be modeled which associate the activation of specific adaptation mechanisms to the activation of the monitors' context-states.

In this scenarion we are interested in the introduction of the following adaptations to the base application described in the paragraph 7.1:

- whenever the user selects a restaurant form the system's gui a path starting from the user's current position and the selected restaurant has to be displayed;
- whenever the user selects a restaurant and there is not enough time for a lunch before the user's flight an alert message is displayed;
- as soon as the user has already checked in for a flight, it is lunch time and there is enough time for a lunch the system automatically displays a message proposing the user to have a lunch in the airport's restaurants which are close to his/her cuisine preferences.

Figure 86 depicts an example of context adapter named *UserLocationAdapter* modeled through the CAMEL editor. It contains ad adaptation mechanisms aimed at modifying the application behavior depending on the user's location. It consists of an adaptation layer named *locateRestaurantLayer* which is triggered by the activation of the *locateRestaurant* context state belonging to the *UserMonitor* context monitor. The *locateRestaurantLayer* consists itself of a behavioral insert named *guiDisplayPathInsert* (Figure 86) which consists of the addition of a method named *displayPath* to the *AeroGui* instance (the *InstanceReference* attribute is setted) passed by the adapter. This method takes as parameters a *startPosition* followed by an *endPosition* and is supposed to displays a suitable path to reach the *endPosition* from the given *startPosition* inside the airport. The behavioral insert only specifies the signagure of the added method but not its implementation which is instead specified throughout the binding named *guiDisplayPath* (Figure 88).

This binding associate an independently defined activity diagram named *diplayPath* (Figure 90) to the *displayPath* operation introduced by the behavioral insert.

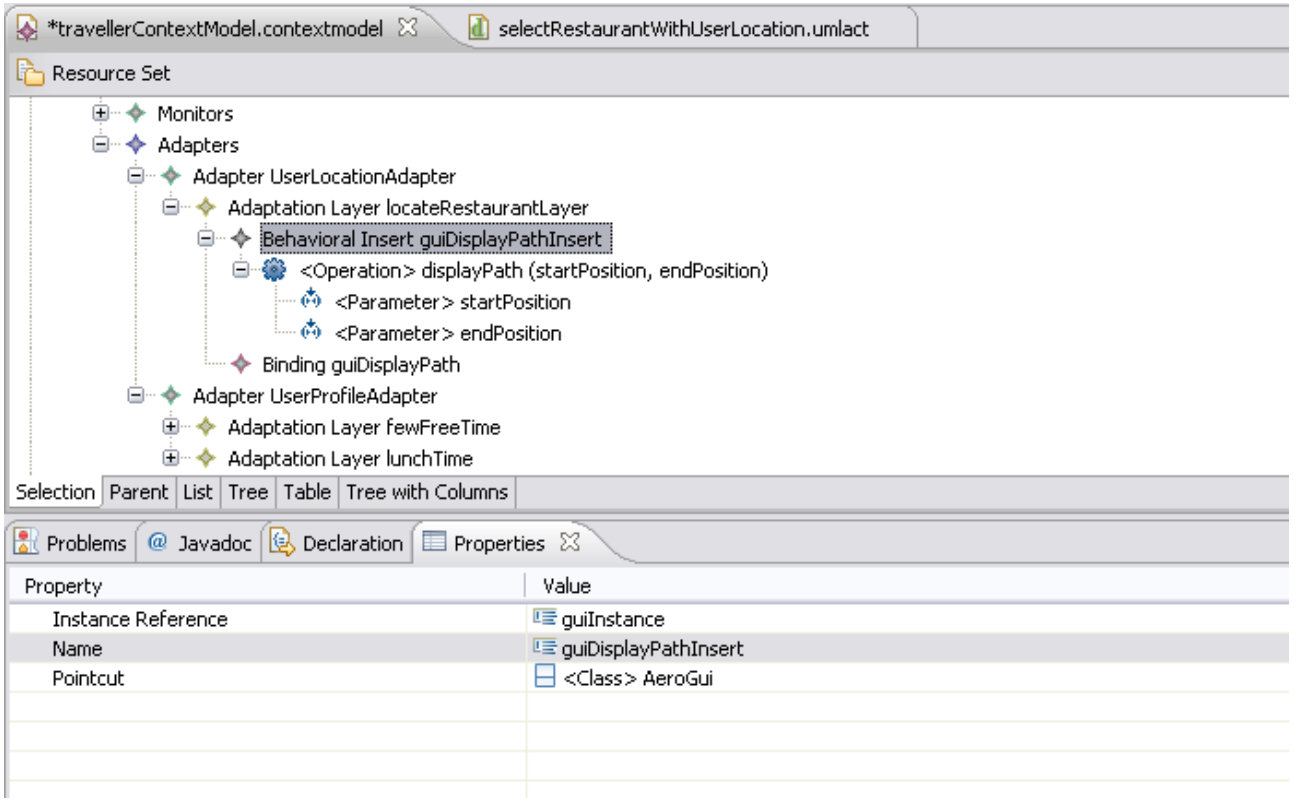

Figure 86: the UserLocationAdapter Context Adapter

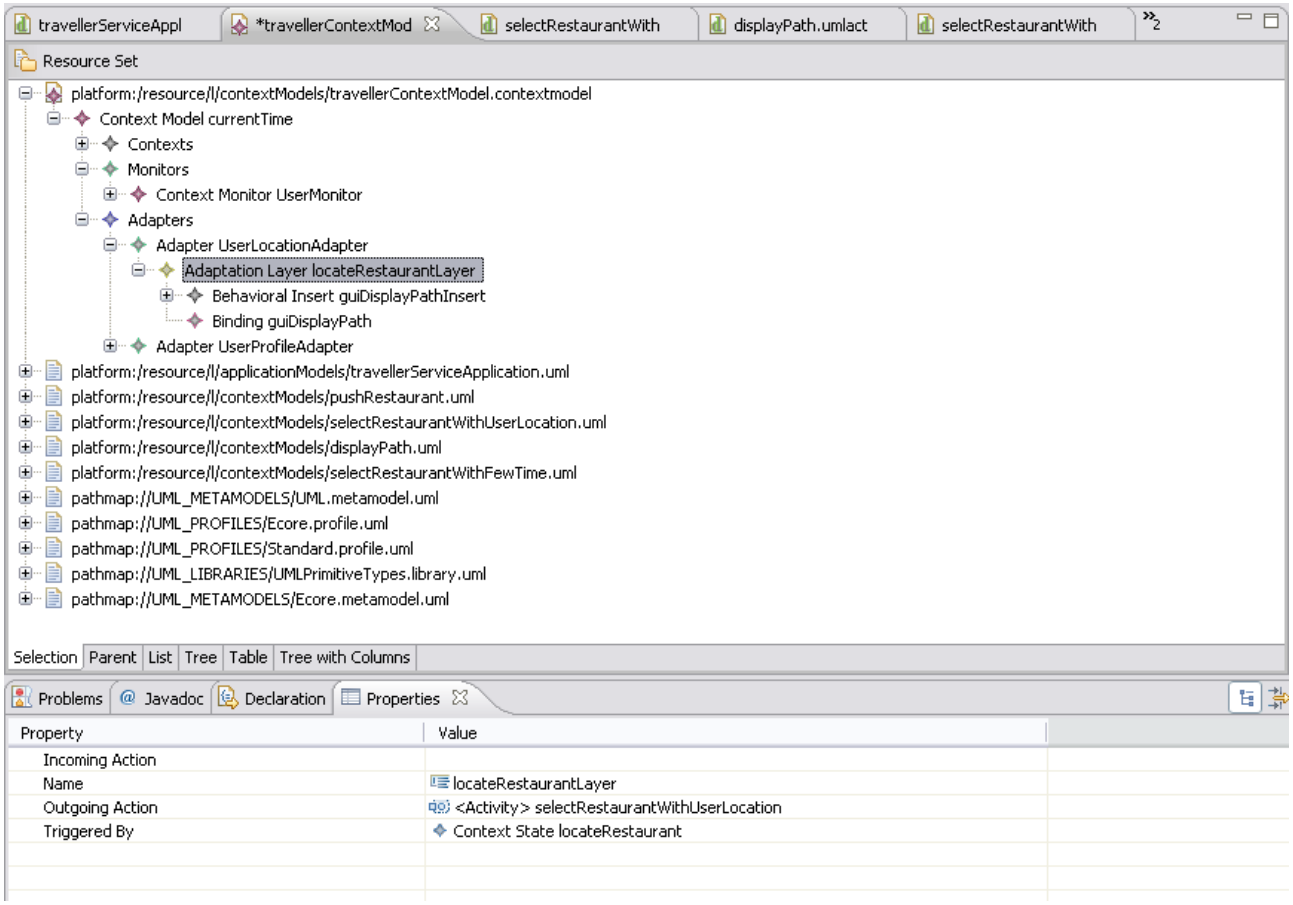

Figure 87: The locateRestaurantLayer Adaptation Layer

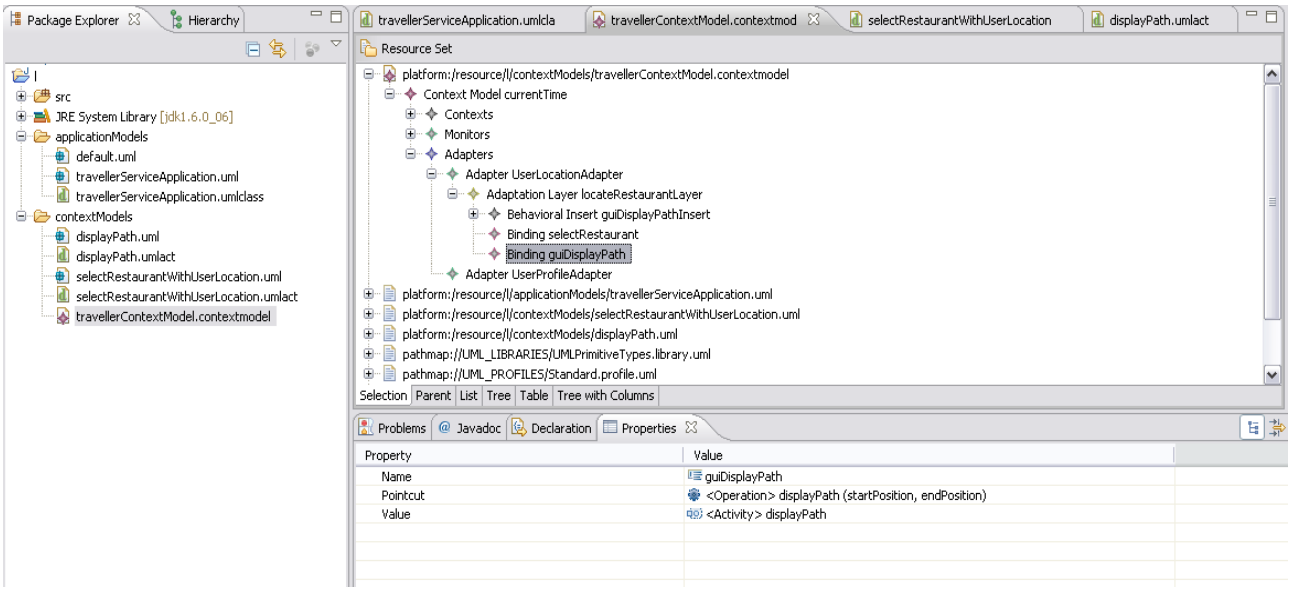

Figure 88: The guiDisplayPath Binding

Figure 87 depicts the CAMEL model of the *locateRestaurantLayer* adaptation layer where an *Outgoing Action* is defined by means of an externally modeled activity diagram named *selectRestaurantWithUserLocation*. The model of this activity diagram is depicted in Figure 89. It takes as input parameter a reference to a restaurant whose actual value is retrieved throughout the *selectedRestaurant* context-attribute exposed by the *locateRestaurant* context-state that has fired the activation of this adaptation layer (Figure 82). This diagram implements the following behavior: it retrieves the user's current position exploiting the *blueToothManager* association and displays a path from the user's current position to the selected restaurant exploting the *displayPath* method added to the *guiInstance* by the adaptation layer itself. Because it is an outgoing activity it is automatically executed whenever the user exits from the *locateRestaurant* context-state that is to say any time he/she has selected a restaurant from the gui.

Figure 91 depicts the model of another adaptation layer named *UserProfileAdapter* containing adaptation mechanisms aimed at modifying the application behavior depending on the user profile. It consists of two adaptation layers respectively named *fewFreeTime* and *lunchtime*. The former is depicted by the Figure 91 and Figure 92, it is aimed to alert the user when he/she selects a restaurant and he/she has less than an hour to the flight to which he/she has checked in. It consists of a behavioral insert named *displayPathInsert* and a Binding named *selectRestaurantBinding*. The former adds

to the *AeroGui* class a method named *diplayProposal* that, when invoked, depicts a message window containing the text passed as parameter and two buttons to let the user makes a boolean choice. The method exits when the user makes a choice returning chosen value. The *selectRestaurantBinding* binding substitutes the default implementation of the *selectRestaurant* method with another implementation provided by the externally modeled activity diagram named *selectRestaurantWithFewTime* (Figure 93) which display a message to the user alerting him/her he/she probably has not enough time for a lunch. If the user chooses to continue the default implementation of the selectedRestaurant method is executed through the *proceed* special construct otherwise the method returns with no further actions. Figure 94 finally depicts the model of the lunchTime adaptation layer which is aimed at proposing the user to have a lunch in those airport restaurants that are close to the user's cuisine preferences. The layer's activation is triggered by the *lunchTime* context-state which is fired when it is approximately lunch time and user has at least one hour of free time before the flight. The layer consists of the same behavioral insert defined for the *fewFreeTime* layer which introduces the *displayProposal* method to the *AeroGui* class and an *Incoming Action* which is modeled by the activity diagram named *pushRestaurant* depicted in Figure 95. The modeled action is fired as soon as the context-state is activated and displays a message proposing the user to have a lunch in the airport's restaurant which are close to its cuisine preferences. If the user accepts the request a list of the available restaurant is retrieved and depicte to the user otherwise the action terminates with no further activities.

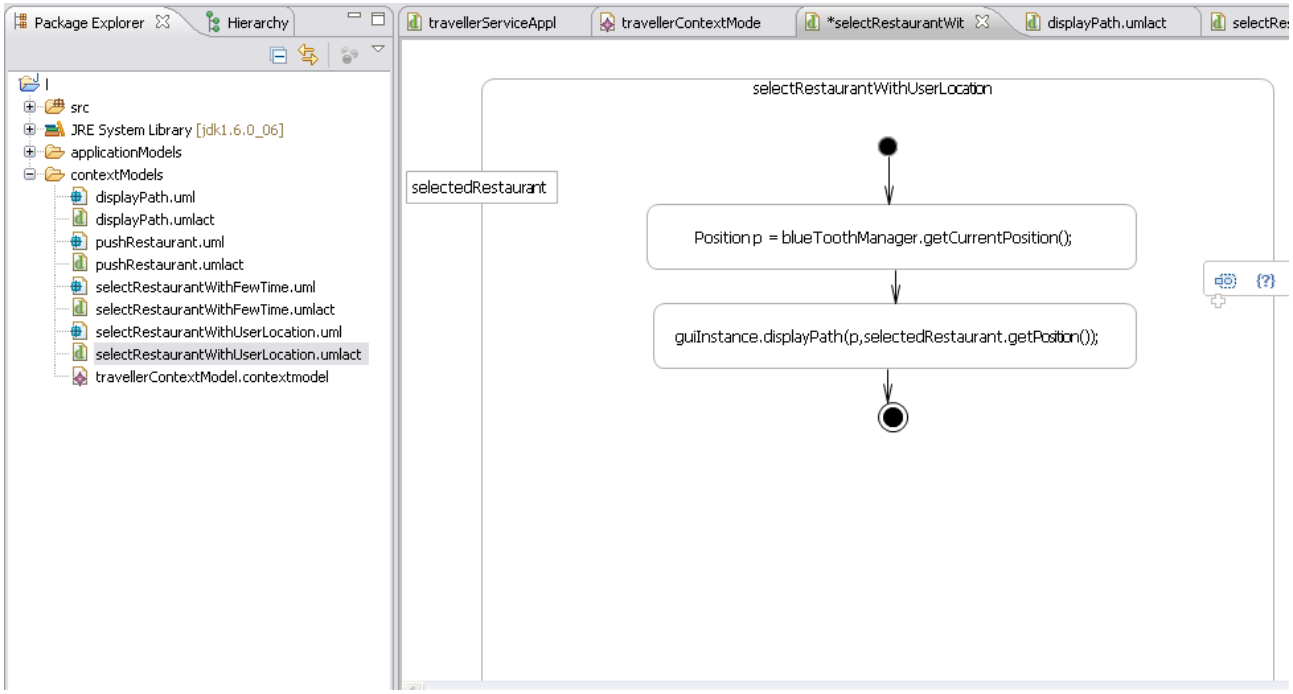

Figure 89: selectRestaurantWithUserLocation activity diagram

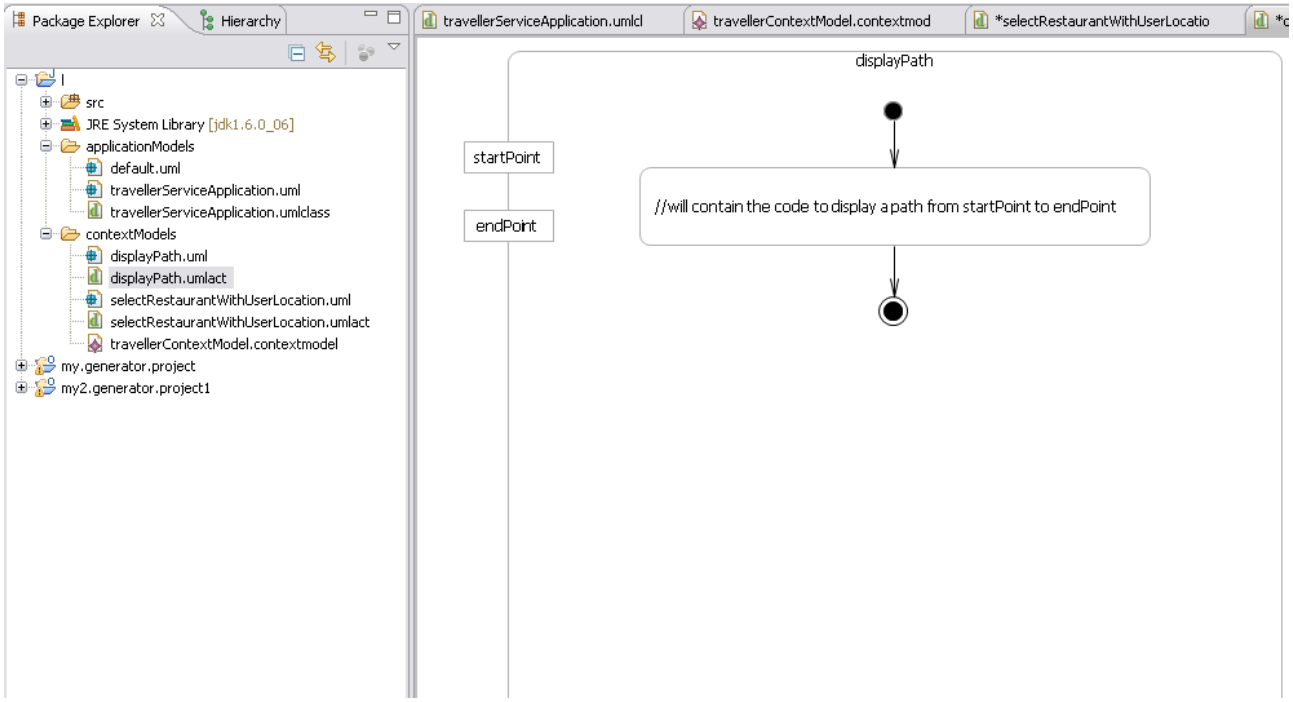

Figure 90: displayPath activity diagram

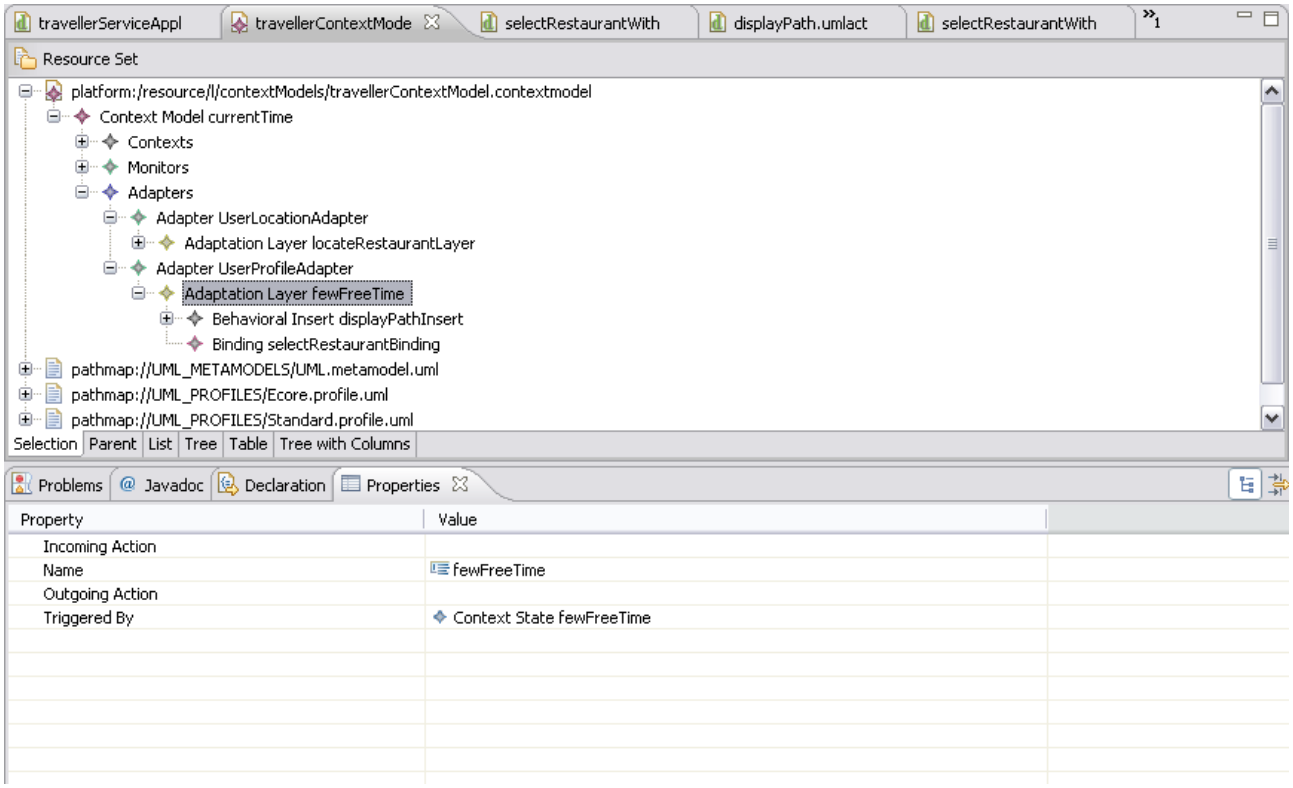

Figure 91: The fewFreeTime Adaptation Layer

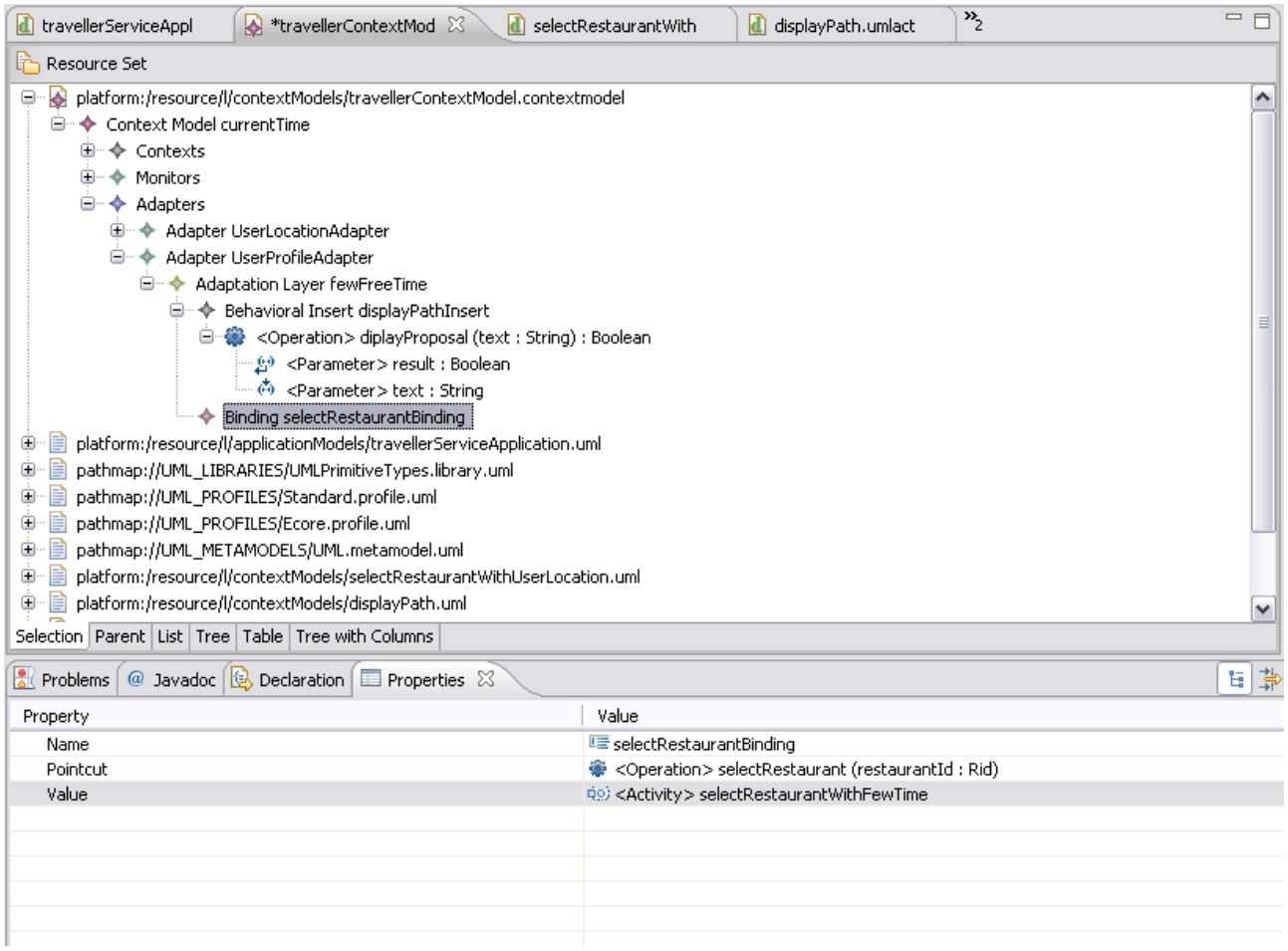

Figure 92: The displayPathInser behavioural insert

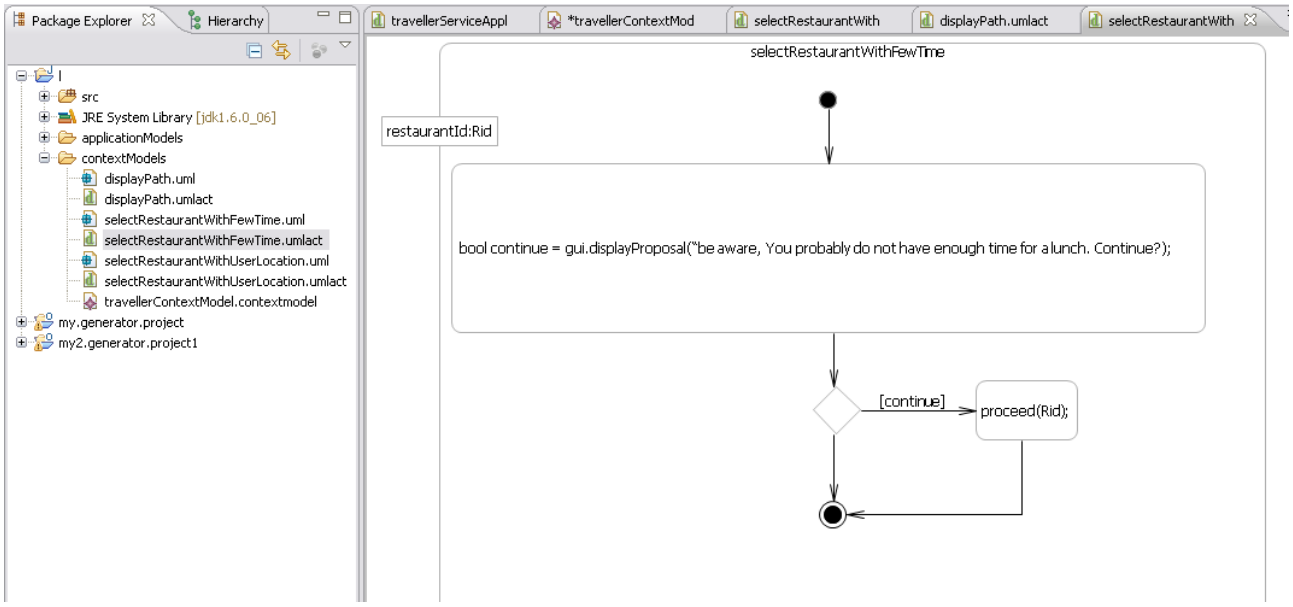

Figure 93: The selectRestaurantWithFewTime activity diagram

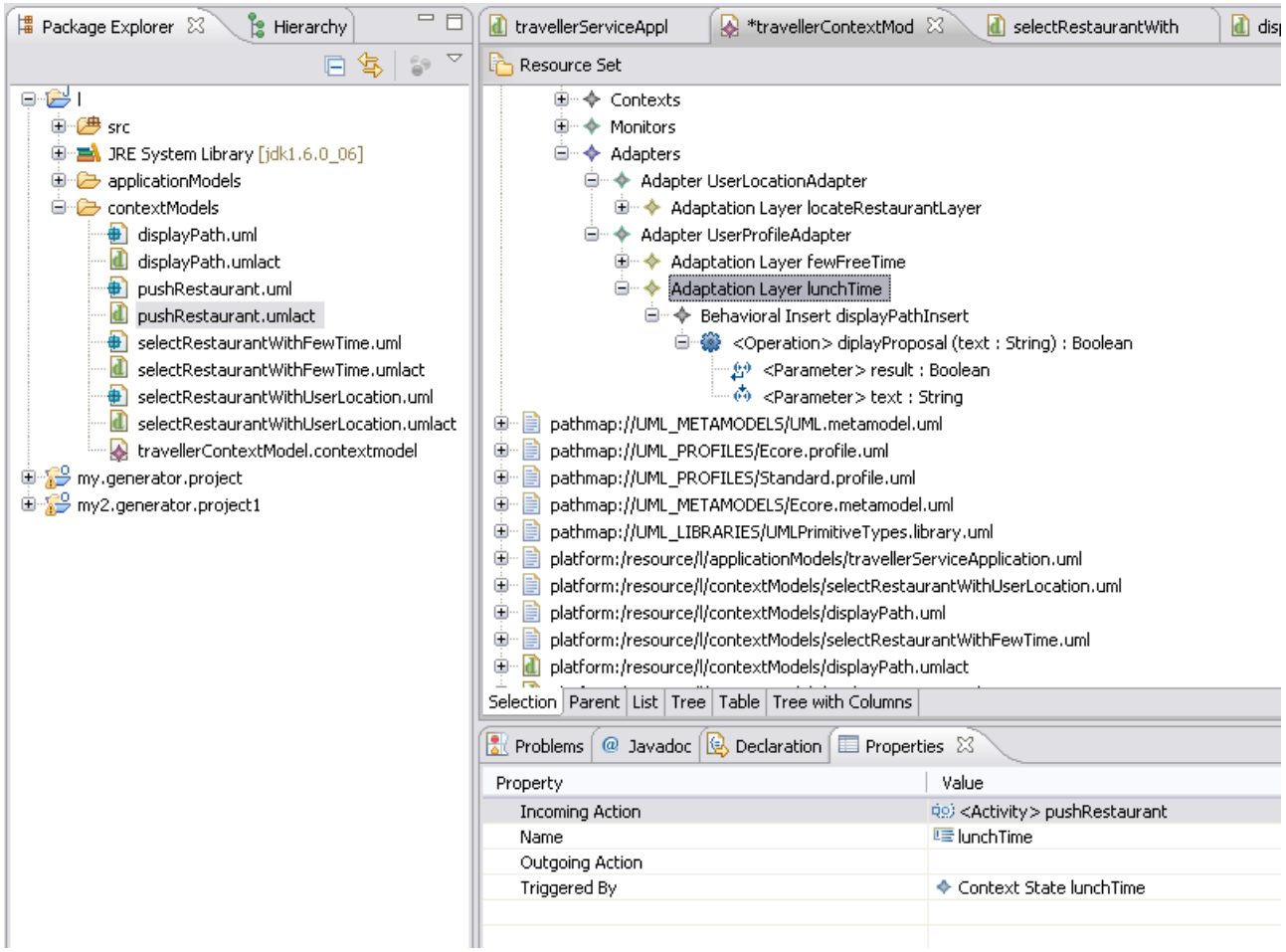

Figure 94: the lunchtime adaptation layer

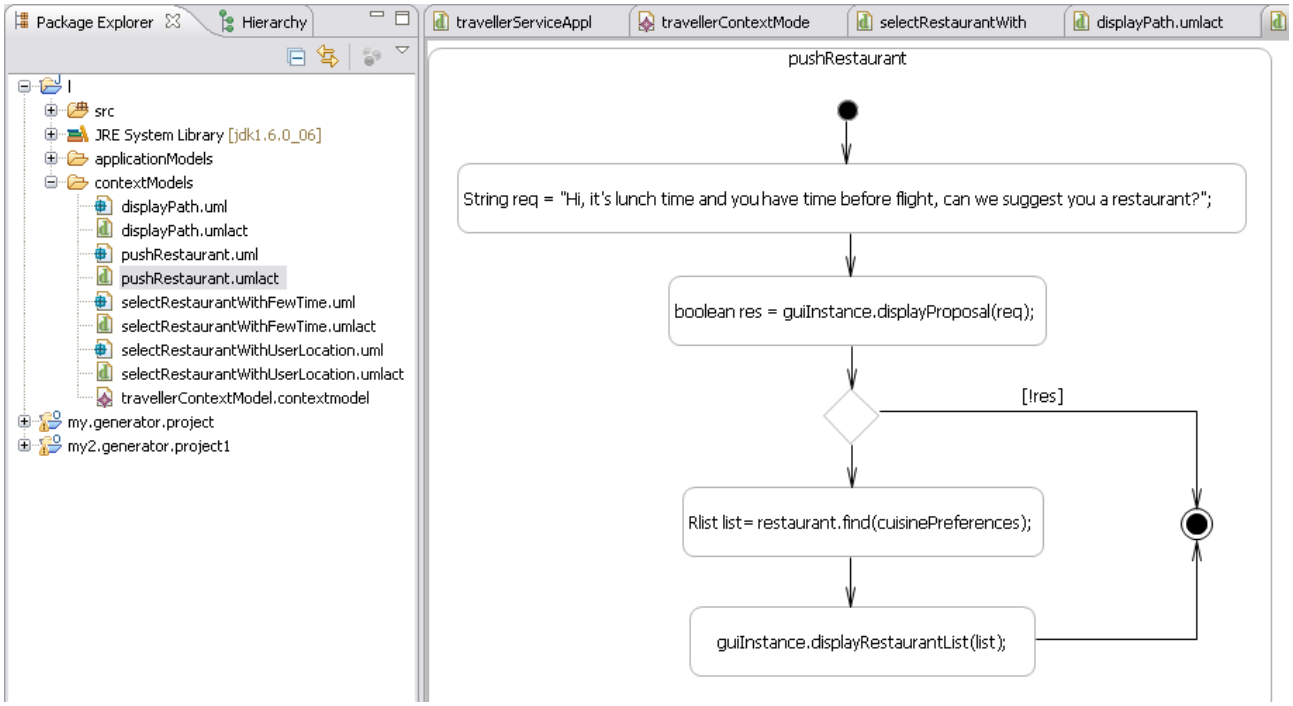

Figure 95: The pushRestaurant activity diagram

7.5 The JCOOL code

In this chapter we propose a possible JCOOL implementation of the CAMEL models defined in the chapter above. Figure 96 depicts the definition of the three modeled contexts, namely: *Time*, *UserProfile* and *UserActivities*. The *Time* context exposes the current time to the context monitors retrieving this information from the *Time* singleton class. The *UserPorfile* context instead collects information about the user retrieving them from the *UserInformation* and Ticket *instances*. The *UserActivities* context finally provides aliases for the *AeroGui::selectRestaurant* and *And AeroGui::checkIn* methods also exposing references to the specific restaurant and the used gui instance.

Figure 97 depicts the definition of the modeled context monitor. It is named *UserMonitor* and *observes* the previously defined context in order to detect three possible context states, namely: *locateRestaurant*, *fewTime* and *lunchTime*.

```
Context Time involves Time {
      currentTime: Time *.currentTime;
}

Context UserProfile involves UserInformation, Ticket{
      completeName: UserInformation *.getCompleteName();

      isCheckedIn: boolean isCheckedIn(){
                return UserInformation *.ownedTicket.isCheckedIn();
      }
      flightTime: int flightTime() {
                return UserInformation *.ownedTicket.getFlightTime();
      }
}

Context UserActivities involves AeroGui {
      locatingRestaurant(Object res, Object guiInstance):
                      AeroGui guiInstance.selectRestaurant(Object res);
      checkingIn(): AeroGui *.checkIn(..);
}
```

Figure 96: Contexts definition

```
Monitor UserMonitor observes Time,UserProfile,UserActivities {
      locateRestaurant(Object res, Object guiInstance):-
                       (locatingRestaurant(res,guiInstance));
      fewFreeTime:- (UserProfile *.isCheckedIn = true),
                    ((Time *.currentTime) >
                       (UserProfile *.flightTime -1:00));
      lunchTime:- (UserProfile *.isCheckedIn = true),
                  ((Time *.currentTime) < 14:00),
                     ((Time *.currentTime) > 12:00),
                    ((Time *.currentTime) <
                      (UserProfile *.flightTime -1:00));
}
```

Figure 97: Monitors definition

The *locateRestaurant* context state goes active as soon as the *AeroGui::selectRestaurant* method is invoked, thus when the user select a restaurant from the GUI. The context state also exposes a reference to the selected restaurant and the exploited *AeroGui* instance. Similarly to the CAMEL model the *fewFreeTime* context is activated as soon as the user has successfully checked in and there is less than one hour to the flight, while the *lunchTime* context state is activated when the user has checked in, it is about lunch and it is more than one hour to the flight.

Figure 98 depicts the JCOOL code implementing the *UserLocationAdapter* adapter, which is responsible to add the displaying of a possible path between the user and a chosen restaurant. To this end the adapter is triggered at any *locateRestaurant* context state activation and exploits a previously defined layer, named *LocationLayer*, that add a new method named *displayPath* to the AeroGui instance which is respectively used to display the path between the user current position and the chosen restaurant position.

Figure 99 finally depicts a possible implementation of the modeled *UserProfileAdapter* which is responsible to alert the user when he/she looks for a restaurant and there is less than an hour to the flight he/she has checked in. In order to do that the adapter applies a layer providing a new implementation for the original *AeroGui::selectRestaurant* method as soon as the *UserMonitor* detects the activation of a *fewFreeTime* context state.

```
AdaptationLayer LocationLayer involves AeroGui {
      public boolean isApplicable(AeroGui instance){
            return true;
      }

      //behavioural inserts
      public void AeroGui *.displayPath(Position start, Position end){
                  //displays a path from start to end
      }

      public void  AeroGui *.displayPath(Restaurant res){
            Position currentPos =
                        BlueToothManager.getInstance().getCurrentPos();
            guiInstance.displayPath(currentPos,
                        restaurant.getPosition());
            }
  }

Adapter UserLocationAdapter drivenBy UserMonitor applies LocationLayer
{
      locateRestaurant(Object res, Object guiInstance){

            //Layers application
            LocationLayer layer = new LocationLayer();
            if(layer.isApplicable(guiInstance)){
                  layer.apply(guiInstance);

            }

            //Incoming and outgoing activities
            in:{
            }
            out:{
                  guiInstance.displayPath(res);
            }

      }

}
```

Figure 98: User location adaptation

```
AdaptationLayer FewTimeLayer involves AeroGui {
          public boolean isApplicable(AeroGui instance){
               return true;
          }

          public boolean AeroGui *.displayProposal(String proposal){
              //displays the proposal passed as parameters
              //giving to the user a boolean choice
          }

          public void AeroGui *.selectRestaurant(Object
          restaurantId){
               bool continue = gui.displayProposal("
                    be aware you probably do not have enough time
                    for a lunch. Would you Continue?);
               if(continue){
                    proceed(restaurantaId);
               }
          }
  }


Adapter UserProfileAdapter drivenBy UserMonitor applies FewTimeLayer {

     fewFreeTime(){
          //Layers application
          FewTimeLayer layer = new FewTimeLayer();
          layer.apply();

          //Incoming and outgoing activities
          int:{}
          out:{}

     }
}
```

Figure 99: User profile adapter

8 Conclusions and Future Works

Context awareness is becoming a key feature for the outcoming pervasive and ubiquitous applications. This trend is leading software engineers to face new challenges in the design of software systems that needs to adapt their structural and behavioral characteristics dependently of the continuous changes of the user's environment and needs.

Difficulties in this field can be summarized by the following bullets:

- Context awareness is a crosccuting concern. It is usually impossible to encapsulate context aware behaviours of a system into a single component;
- There is not a common agreement on what is a context, what is the definition of *context aware*;
- There is not a common agreement on how context and context aware behaviors should be modeled and then implemented.

Several approaches have been proposed to address the open issues that may arise during the design phase of a context aware system, with context oriented modeling [5,6,7, 8,9,10], or at the coding phase with context oriented programming [11,12,13,14,15]. In order to provide a contribution toward a better understanding of context awareness in software engineering, in this thesis we have presented a modeling framework that can be exploited by designers to model systems' context aware characteristics independently of the the way they could be implemented but focusing on the information related to the involved entities.

The most important features the developed framework provides are:

- the possibility for the designer to quickly develop models capturing his/her understanding of the context awareness concerns possibly making easier to share them with the other system's stakeholders;
- The possibility to model the intruduction of context-aware capabilities into an already existing system avoiding to modify its existing models or code;
- The possibility to apply model transformations aimed at automatically producing artifacts (code, metrics, etc.) from the defined models.

For the realization of such framework we first started with a comparison of the already existing context awareness modeling and development approaches aimed at identifying the common characteristics and most expressive solutions that have already been proposed in literature. This lead us to the definition of a conceptual domain model

(CDM) which tries to take the best form the already proposed approaches. This model consists of a taxonomy of concepts representing the main concerns of involved context awareness and is aimed at providing a common vocabulary to which existing approaches can be related and by which new approaches can be instantiated. In the 3rd chapter we describe the designed CDM for context awareness also providing a brief comparison between it and the approaches described in the 2nd chapter. This brief comparison is aimed at providing an informal demonstration of how our CDM can encompass the existing approaches possibly becoming a common vocabulary to which they can be related and by which new approaches can be instantiated. Moreover, because of its abstract nature it could also become an instrument of models interchange between different approaches. From the designed CDM we have then defined a domain specific modeling language (DSML) for context awareness modeling we have called CAMEL (Context Awareness ModEling Languge). CAMEL has been defined as a heavy weight UML extension that makes possible to introduce context aware characteristic into already UML modeled systems without modifying the existing models. Exploiting the Eclipse platform and the Eclipse Modeling Framework an editor for CAMEL models has been automatically generated from a formal representation of the CAMEL meta-model. The obtained editor represents the core component of a modeling framework for context awareness that makes possible to apply model transformations aimed at producing:

- metrics or other measurments giving the designer feedbacks about his/her desigin choices;
- documentation which can be shared with the system's stakeholders;
- code which actually implements them;
- other artifacts;

starting from well formed CAMEL models.

In the 4th and 5th we have then described the Model Driven Development paradigm together with a brief description of the Eclipse [19] and the Eclipse Modeling Framework (EMF) [20] which is the technological platform we have chosen for the development of our environment and the development process of the CAMEL editor itself.

In the 6th chapter we have finally introduced the code level counterpart of our approach consisting of a domain specific language called JCOOL (Java Context Oriented Language)[21] that is a Java extension explicitly designed for context oriented purposes. JCOOL represents the code level implementation our conceptual domain model. It has been presented in [21] for the first time but then consistently modified in order to

improve its expressiveness. JCOOL has to be considered only one of the possible target platforms into which CAMEL models can be automatically transformed by means of suitable transformation processes. Because of the platform independency of the CAMEL models several other transformations with respect to different target platforms are infact possible (i.e., ContextJ combined with ContextToolkit, AspectJ, Jasco, etc.).

In the 7th chapter we have finally proposed a simple application scenario to give a concrete example of how the implemented framework can be exploited to introduce context awareness characteristics into a, possibly already existing, independently designed application. To this end the used scenario is the traveler service application scenario introduced in [3]. Starting from a brief specification of this application, we have shown how the Eclipse UML plugin can be exploited to produce a proper UML model of the base application. The target system model can then be referenced by CAMEL models defined with our modelling framework in order to introduce the desired context aware features. These models can then act as input of model transformation workflows whose final purpose is to produce executable code or other desirable artefacts (i.e. documentation, metrics, etc.).

What presented in this thesis represents the first step of an ongoing work which crosscuts different branches of software engineering such as *meta-modeling*, *domain specific languages*, context-awareness, *aspect-orientation* and so on.

In the close future we are about to produce:

- a stable version of the realized CAMEL plugin ecompassing a graphical user interface alternative to the already existing tree based gui;
- an extension of the implemented JCOOL parser with the capability to automatically generate AspectJ code realizing well formed JCOOL programs;

Further steps will also include:

- The modeling and development of distributed context aware characteristics. This task include the sensing of context-information from distributed sensors and the definition and activation of adaptation layers which affect distributed components;
- The modeling and development of knowledge based contextual inference. Inference is a deduction mechanism that can be exploited In order to deduce new contextual information of an higher level of abstraction with respect to the currently known.

The defined CAMEL and JCOOL languages are specifically tailored for the modeling and implementation of components based object oriented systems. In the future we would like to investigate about the possibility to define CAMEL and JCOOL extensions for the modeling of context awareness in Web Services based distributed systems.

Bibliography

1. M. Satyanarayanan, “Pervasive Computing: Vision and Challenges,” Personal Comm., Vol. 8, no. 4, Aug. 2001, pp. 10-17;
2. M. Weiser, “The Computer for the 21st Century,” Scientific American, September 1991
3. V. Grassi, A. Sindico, “Towards Model Driven Design of Service Based Context Aware Applications,” In the Proceedings of the International Workshop on Engineering of software services for pervasive environments (ESEC/FSE’07), Dubrovnik, Croatia, 2007;
4. V. Grassi, A. Sindico, “UML Modeling of Static and Dynamic Aspects,” 9th International Workshop on Aspect Oriented Modeling (AOM), Genova, Italy, October 2006;
5. T. Strang, L. Linnhoff-Popien, “A Context Modeling Survey,” in 1st International Workshop on Advanced Context Modeling, Reasoning and managament, Nottingham, 6th International Conference on Ubiquitous Computing, UK, pp 33-40;
6. B. N. Shilit, N. L. Adams, R. Want, “Context-aware computing applications,” in IEEE Workshop on Mobile Computing Systems and Applications (Santa Cruz, CA, US, 1994);
7. Q. Z. Sheng, B. Benatallah, “ContextUML: a UML-based modeling language for model-driven development of context aware web services,” IEEE Int. Conf. on Mobile Business (ICMB’05).2005;
8. J. Bauer, “Identification and Modeling of Contexts for Different Information Scenarios in Air Trafic, “ Mar. 2003, Diplomarbeit;
9. K. Henricksen, J. Indulska, A. Rakatonirainy , “A Generating Context Management Infrastructure from High-Level Context Models,” In Industrial Track Proceedings of the 4th International Conference on Mobile Data Management (MDM2003), Melbourne, Australia, January 2003, pp. 1-6;
10. K. Henricksen, J. Indulska, A. Rakotonirainy, “Modeling Context Information in Pervasive Computing Systems,” In Proceedings 1st Internation Conference on Pervasive Computing 2414, pp. 79-117, Zurich;
11. A. Rakotonirainy, “Context-Oriented Programming for Pervasive Systems,” Technical Report, University of Queensland, September 2002;

12. P. Costanza, R. Hirchfeld, “Language Constructs for Context-Oriented Programming: An Overview of Context L.” In: Proceedings of the Dynamic Languages Symposium (DLS)’05, co-organized with OOPSLA’05, New York, NY, USA, ACM Press (October 2005);
13. M. Gassanenko, “Context Oriented Programming: Evolution of Vocabularies,” Proceedings of the euroFORTH’93 Conference. Marianske Lazne, Czech Republic;
14. M. Gassanenko, “Context-oriented Programming,” euroFORTH’98, Schloss Dagstuhl, Germany;
15. R. Keays, A. Rakatonirainy, “Context-oriented Programming,” International Workshop on Data Engineering for Wireless and Mobile Access, San Diego, Usa, 2003. ACM Press;
16. F. Lagarde, H. Espinoza, F. Terrier, S. Gerard, “Improving UML Profile Design Practices by Leaveraging Conceptual Domain Models,” In the Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. 2007, ISBN:978-1-59593-882-4;
17. B. Selic, “A Sistematic Approach to Domain-Specific Language Design Using UML,” 10th IEEE ISORC ’07, 2007;
18. OMG – The Object Management Group: http://www.omg.org/;
19. Eclipse – www.eclipse.org;
20. Eclipse Modeling Framework (EMF) – http://www.eclipse.org/modeling/emf/;
21. A. Sindico, G. Bartolomeo, V. Grassi, S. Salsano, “Design and Development of a Context Oriented Language for Middleware Based Applications,” workshop on Next Generation Aspect Oriented Middleware (NAOMI’08), Seventh International Conference on Aspect-Oriented Software Development (AOSD 2008), Brussels, Belgium;
22. AspectJ – www.eclipse.org/aspectj;
23. W. Vanderperren, D. Suvee, M. Cibran, B. Verheecke, V. Jonckers, ”Adaptive Programming in JasCo,” in Proceedings of AOSD 2005. ACM Press, Chicago, USA;
24. Special Issue of Human-Computer Interaction, Volume 16, 2001;
25. D. A. Norman, “The Invisible Computer,” MIT Press, Cambdirge, MA, 1998;
26. T. Moran, “Introduction to This Special Issue on Context in Design,” In Human-Computer Interactions, 9(1) pp.1-2;
27. Apple iPhone, http://www.apple.com/iphone/;
28. Nintendo Wii, http://www.nintendo.com/wii/;

29. J. C. Lee Wii Remote Projects: http://ww.cs.cmu.edu/~johnny/projects/wii/;
30. B. Shilit, M. Theimer, "Disseminating active map information to mobile hosts," IEEE Network 1994; 8; 22-32;
31. R. Want, A. Hopper, V. Falcao, J. Gibbons, "The Active Badge Location System," ACM Transactions on Information Systems (TOIS),v.10 n.1, p.91-102, Jan. 1992;
32. A. K. Dey, G. D. Abbowd, "Towards a Better Understanding of Context-Awareness," Proceedings of the Workshop on the What, Who, Where and How of Context Awareness, affiliated with the CHI2000 Conference on Human Factors in Computer System, New York, NY: ACM Press;
33. N. Ryan, J. Pascoe, D. Morse, "Enhanced Reality Fieldwork: the context Aware Archaeological Assistant", Computer Applications in Archaology (1997);
34. B. Schilit, M. Theimer, "Disseminating Active Map Information to Mobile Hosts", IEEE Network, 8(5)(1994);
35. P. J. Brown, J. D. Bovey, X. Chen, "Context-Aware Applications: From the Laboratory to the Marketplace," IEEE Personal Communications, 4(5)(1997)58-64;
36. A. K. Dey, "Context Aware Computing: The CyberDesk Project," AAAI 1998 Spring Symposium on Intelligent Environments, Technical Report SS-98-02 (1998) 51-54;
37. B. Shilit, N. Adams, R. Want, "Context Aware Computer Applications," 1th International Workshop on Mobile Computing Systems and Applications. (1994) 85-90;
38. J. Pascoe, "Adding Generic Contextual Capabilities to Wearable Computers," 2nd International Symposium on Wearable Computers (1998)92-99;
39. R.Kernchen, M. Boussard, C. Hesselman, C. Villalonga, E. Clavier, A. V. Zhdanova, P. Cesar, "Managing Personal Communication Environments in Next Generation Service Platforms," IST Mobile and Wireless Communications Summit, Budapest, 2007, pp. 1-5;
40. R. Hull , P. Neaves, J. Bedford-Roberts, "Towards Situated Computing," 1st International Symposium on Wearable Computers (1997)146-153;
41. J. Pascoe "Adding Generic Contextual Capabilities to Wearable Computers," 2nd International Symposium on Wearable Computers (1998) 92-99;
42. J. Pascoe, N. S. Ryan, D. R. Morse, "Human-Computer-Giraffe Interaction – HCI in the Field," Workshop on Human Computer Interaction with Mobile Devices (1998);

43. B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, "Software Engineering for Self-Adaptive Systems," Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany 2008;
44. B.H.C. Cheng, J. M. Atlee, "Research directions in Requirements Engineering," in Future of Software Engineering 2007 (FOSE '07), pages 285-303. IEEE Computer Society, Minneapolis, MN, USA, May 2007;
45. E. Chtcherbina M. Franz, "Peer-to-peer coordination framework (p2pc): Enabler of mobile and-hoc networking for medicine, business and entertainment," In Proceedings of International Conference on Advances in Inrastructure for Electronic Business, Education, Science, Medicine and Mobile Technologies on the Internet (SSGRR2003w), L'Aquila (Italy), January, 2003;
46. A. Schmidt, M. Beigl, H. Gellersen, "There is More Context than Location," In Computers and Graphics, 23, 6, (1999), 893-901;
47. J. McCarthy, "Notes on formalizing contexts," In Proceedings of the Thirteen International Joint Conference on Artificial Intelligence (San Mateo, California, 1993), R. Bajcsy, Ed. Morgan Kaufmann, pp. 555-560;
48. M. Uschold, M. Gruninger, "Ontologies: Principle, methods, and applications," Knowledge Engineering Review 11, 2 (1996), 93-155;
49. T. G. Gruber, "A Translation approach to portable ontologies," Knowledge Acquisition 5, 2 (1993), 199-200;
50. A. Shehzad, H. Q. Ngo, K. Anh Pham, S. Y. Lee, "Formal Modeling in Context Aware Systems," In the Proceedings of The 1st International Workshop on Modeling and Retrieval of Context (MRC '2004), Ulm, Germany, 2004;
51. O. Saidani, S. Nurcan , "Towards Context Aware Business Process Modelling," In the Proceedings of the 8th Workshop on Business Process Modeling, Development and Support (BPMDS'07), Trondheim, Norway, June 2007;
52. D. Rios, P. Dockhorn, et al., "Using Ontologies for modeling Context aware Services Platforms, " In Workshop on How to Use Ontologies and Modularization to Explicitly Describe the Context Model of a Software Systems Architecture, in OOPSLA'03, Anaheim, CA, USA, October 2003;
53. N.Q. Hung, A. Shehzad, S. L. Kiani, M. Riaz, S. Lee, "A Unified Middleware Framework for Context Aware Ubiquitous Computing," In EUC2004, Japan, Aug, 2004;

54. W3C Web Ontology Working Group, “The Web Ontology Language: OWL,” http://www.w3.org/2001/sw/WebOnt/;
55. FORTH Language: http://c2.com/cgi/wiki?ForthLanguage;
56. D. Salber, A. K. Dey, G. D. Abowd, “The Context Toolkit: Aiding the Development of Context-Enabled Applications,” In the Proceedings of CHI’99, May 1999;
57. R. Hirschfeld, P. Costanza, O. Nierstasz, “Context-oriented Programming,” in Journal of Object Technology (JOT), vol 7, no. 3, pages 125-151, March-April 2008, http:// ww.jot.fm;
58. W. Harrison, H. Ossher, “Subject Oriented Programming: a critique of pure objects,” In Proceedings of the eigth annual conference on Object-oriented programming systems, languages and applications, 1993, Washington, D.C., United States;
59. F. Irmert, T. Fischer, K. Meyer-Wegener, “Runtime Adaptation in a Service Oriented Component Model,” In the Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Leipizig, Germany, 2008;
60. F. Irmert, Meyerh¨ofer, M. Weiten, “Towards Runtime Adaptation in a SOA Environment,” In W. Cazzola, S. Chiba, Y. Coady, S. Ducasse, K. Kniesel, M. Oriol and G. Saake, editors, Proceedings of ECOOP’2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE’07), pages 17-26, Berlin, Germany, 2007;
61. G. Kniesel, “Type-safe delegation for run-time component adaptation,” In ECOOP’99: Proceedings of the 13th European Conference on Object-Oriented Programming, pages 351-366, Londong, UK, 1999;
62. A. Popovinci, G. Alonso, T. Gross, “Just in Time Aspects: Efficient Dynamic Weaving for Java,” In AOSD’03: Proceedings of the 2nd International Conference on Aspect Oriented Software Development, pages 100-109, New York, USA, 2003, ACM Press;
63. E. Bruneton, T. Coupaye, M. Leclercq, V. Qu’ema, J. Stefani, “The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems,” Softw. Pract. Exper., 36:1257-1284,2006;
64. T. Bloom, “Dynamic Module Replacement in a Distributed Programming System,” PhD thesis, Massachussets Institute of Technology, 1983;

65. A. Mukhija, M. Glinz, “Runtime adaptation of applications through dynamic recomposition of components,” in Pro. Of 18th International Conference on Architecture of Computing Systems, 2005;
66. J. Kramer, J. Magee, “The Evolving Philosophers Problema: Dynamic Change Management,” IEEE Transcations on Software Engineering, 16(11):1293-1306, 1990;
67. C. Bidan, V. Issarny, T. Saridakis, A. Zarras, ”A Dynamic Reconfiguration Service for CORBA,” 1998;
68. G. Kiczales, J. Lamping, A. Mendhekar et al., ”Aspect Oriented Programming,” in proc. European Conference on Object Oriented Programming. Finland 1997;
69. W. E. Dijkstra, ”On the role of scientific thought,” Selected writings on Computing: A Personal Perspervice, New York, USA: Springer-Verlag New York, pp.60-66 ISBN 0-387-90652-5;
70. E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, ”Design Pattern: Elements of Reusable Object Oriented Software,” Addison-Wesley, Readin, MA, 1995;
71. L. Lengyel, T. Levendovszky, H. Charaf, “Aspect Oriented Techniques in Metamodel-Based Model Transformation,” 6th International symposium of Hungarian Researchers of Computational Intelligence, November 18-19 2005, Budapest;
72. K. Masuhara, G. Kiczales and C. Dutchyn, “Compilation semantic of aspect oriented programs,” in FOAL 2002 Proceedings: Foundation of Aspect-Oriented Languages Workshop at AOSD 2002, G. T. Leavens and R. Cytron, Eds. Tech. 02-06. Department of Computer Science, Iowa State University, 17—26;
73. W. Cazzola, J. M. Jezequel, A. Rashid, “Semantic Join Point Models: Motivations, Notions and Requirements,” In Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT ’06), Bonn, Germany, on 21st March 2006;
74. The UML 2.1.2 specification – http://www.omg.org/spec/UML/2.1.2/;
75. E. Tanter, K. Gybels, M. Denker, A. Bergel, “Context-aware aspects,” In Proceedings of the 5th International Symposium on Software Composition (SC 2006), LNCS, pages 227-249, Vienna, Austria, Mar. 2006, Springer-Verlag;
76. T. Rho, M. Schmatz, A. B. Cremers, “Towards Context-Sensitive Service Aspects,” In Work on Object Technology for Ambient Intelligence and Pervasive Computing, 2006;

77. J. Balasubramanian, B. Natarajan, D. C. Schmidt, A. S. Gokhale, J. Parsons, G. Deng, ”Middleware Support for Dynamic Component Updating,” In OTM Conferences (2), volume 3761 of Lecture Notes in Computer Science, pages 978-986, Springer, 2005;
78. The MOF specification – http://www.omg.org/spec/MOF/2.0;
79. N. Chomsky, “Three models for the description of language,” IRE Transaction on Information Theory IT-2, 113-124 (1956);
80. P. Costanza, R. Hirschfeld, W. De Meuter, “Efficient Layer Activation for Switching Context-dependent Behavior,” Joint Modular Languages Conference 2006 (JMLC2006), Oxford, England, September 13-15, 2006, Springer LNCS;
81. P. Costanza, R. Hirschfeld, “Reflective Layer Activation in ContextL,” ACM Symposium on Applied Computing, Mach 11-15, Seoul, Korea;
82. P. Durr, T. Staijen, L. Bergmans, M. Aksit, ”Reasoning About Semantic Conflicts Between Aspects,” IN EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software;
83. V.S. Waralak, “Ontologies and Object models in Object Oriented Software Engineering,” IAENG International Journal of Computer Science, 33:1;
84. W. Vongdoiwang, D. N. Batanov, “Similarities and Differences between Ontologies and Object Model,” CCT’05 proceeding 2004, Austin, Texas;
85. A. K. Dey, “A Conceptual Framework and Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications,” Anchor article of a special issue on Context Aware Computing, Human-Computer Interaction (HCI) Journal, Vol 16 (2001);
86. J. Mukerji, J. Miller, “Model Driven Architecture,” http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, July 2001;
87. C. Atkinson, T. Kühne, “Model-Driven Development: A Metamodeling Foundation,” IEEE Software, vol. 20, no. 5, pp. 36-41,2003;
88. D. S. Frankel, “Model Driven Architecture: Applying MDA to Enterprise Computing,” OMG Press, 2003;
89. IBM – www.ibm.com;
90. Microsoft – www.microsoft.com;
91. J. Bettin, G. van Emde Boas, “Generative Model Transformer,” In the proceedings of the Object Oriented Programming Systems Languages and Applications, ISBN: 1-58113-751-6 pages 88-89;

92. K. Henricksen, J. Indulska, “Modelling and Using Imperfect Context Information,” In: Pervasive Computing and Communications Workshop, 2004, ISBM: 0-7695-2106-1, pp 33-37;
93. Generic Modeling Environment (GME) - http://www.isis.vanderbilt.edu/projects/gme/;
94. www.planetmde.org
95. ORMSC White Paper, “A Proposal for MDA Foundation,” http://www.omg.org/docs/ormsc/05-04-01.pdf;
96. T Mens, K. Czarnercki, P. Van Gorp, ”A Taxonomy of model transformation,” In Bezivin, Jean; Heckel, Reiko (Hrsg): Language Engineering for Model-Driven Software Development. Dagstuhl Germany 2005;
97. N. Wirth, “Program development by stepwise refinement,” ACM 14 (1971);
98. R. J. Back, J. Von Wright, “Refinement Calculus,” Springer Verlag (1998);
99. MDT: http://www.eclipse.org/modeling/mdt/?project=uml2;
100. S. C. Johnson, “YACC: Yet another compiler compiler,” Tech. Rep. 32. AT&T Bell Laboratories, Inc., Murray Hill, NJ (1975);
101. T. J. Parr, R. W. Quong, “ANTLR: a predicated-LL(k) parser generator,” Software-Practice & Experience, V. 25 n. 7, p. 789-810, July 1995;
102. ANTLR parser generator: http://www.antlr.org;
103. M. O’Neill, C. Ryan, “Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language,” (2003) Kluwer Academic Publishers;
104. K. Henricksen, J. Indulska, “A Software Engineering Framework for Context-Aware Pervaisve Computing”, In: Pervasive Computing and Communications (PERCOM’04), ISBN: 0-7695-2090-1, pp- 77-86;
105. R. B. Smith, D. Ungar, “A Simple and Unifying Approach to Subjective Objects,” TAPOS special issue on Subjectivity in Object-Oriented Systems, 2(3):161-178, 1996;
106. D. Batory, A. Rauschmeyer, “Scaling step-wise refinement. IEEE Transactions on Software Engineering,” June 2004;
107. S. S. Yau, F. Karim, Y. Wang, B. Wang, S. Gupta, “Reconfigurable Context-Sensitive Middleware for Pervasive Computing,” (Jul.-Sep.2002) 33-40;