Received 25 Oct., 2018; | Revised 18 Apr., 2019; | Accepted 19 May, 2019

DOI: xxx/xxxx

ARTICLE TYPE

An MDD-based method for building context-aware applications with high reusability

Nearchos Paspallis*

[1]School of Sciences, UCLan Cyprus, Larnaca, Cyprus

Correspondence
*Nearchos Paspallis, 12-14 University Avenue, Pyla, CY-7080, Cyprus. Email: npaspallis@uclan.ac.uk

Present Address
12-14 University Avenue, Pyla, CY-7080, Cyprus

Abstract

Adding context-awareness capabilities to modern mobile and pervasive computing applications is becoming a mainstream activity in the software engineering community. In this respect, many context models and middleware architectures have been proposed with the aim to provide the developers with tools and abstractions that make it easier to produce context-aware applications. However, current solutions suffer from relatively low reusability and lack ease-of-use. In this paper, we propose a two-layer approach based on model-driven development: at the higher layer we introduce the design of reusable context plug-ins which can be used to monitor low-level context data and to infer higher-level information about the users, their computing infrastructure and their interaction. At the lower layer, the plug-ins themselves are synthesized using more elementary, reusable components. We argue that this development approach provides significant advantages to the developers, as it enables them to design, implement, re-use and maintain the code-base of context-aware apps more efficiently. To evaluate this approach, we demonstrate it in the context of a two-part case-study and assess it both qualitatively and quantitatively.

KEYWORDS:
Context-awareness; MDD; Reusability; CBSE; Middleware; Plug-in; Code generation

1 | INTRODUCTION

Context-awareness is one of the most sought-after features for modern mobile and pervasive computing applications. Understanding and using context allows applications to identify and predict *intention* thus allowing for better and richer human-computer interaction.

Researchers have proposed various solutions to ease the task of designing and implementing applications that are context-aware and self-adaptive. At their core many such approaches build on reusable code, ranging from the early works of Dey[1] to more recent, plug-in based architectures such as those described in [2,3,4,5]. These frameworks are designed to allow the applications which are using them to inquire and access arbitrary context data types. In this respect, the context-awareness and context-inferring parts are separated from the functional logic of the applications by means of reusable *context plug-ins* (i.e., individually deployable units which provide mechanisms for collecting and processing context data, and inferring higher-level context information). At the same time, many context models and middleware architectures were proposed and documented in the literature [6,7]. Software reuse is at the core of many of those approaches. At a higher level, reuse covers all software engineering phases and "*can be supported by different types of methods, including ad hoc, opportunistic, adaptive, systematic, black-box, whitebox, etc.*"[8].

A common drawback in these approaches is that the developers are required to invest a significant amount of time to develop customized components which are used for collecting, processing, inferring, storing, querying and accessing context data. Moreover, the developed components are then harder to reuse and more prone to errors, preventing cost-effective development. We argue that an approach which uses component-based development in combination with a component repository can greatly facilitate the development of such context-aware applications, making the development of highly capable and robust context-aware applications more affordable.

In this paper we propose a *Model-Driven Development* (MDD) approach to facilitate easy and automated development of reusable context plug-ins, and as a result to enable context-awareness features in the developed applications. The proposed approach includes the specification and implementation of context plug-ins via finer, reusable components, assembled using an MDD tool-chain. This approach is validated in the context of a case study application, qualitatively evaluated against related work and quantitatively assessed in terms of development time and resulting complexity.

The main contributions of this paper can be summarized as follows:

- Specification of a comprehensive, *Model Driven Development*-based approach for building reusable context plug-ins.
- A methodology which enables the formation of context plug-ins out of finer, reusable components.
- The definition, implementation and analysis of two extensive case study scenarios, demonstrating and evaluating the development methodology.

The rest of this paper is organized as follows. Section 2 provides an overview of the underlying context architecture, with emphasis on its model, context query language and pluggable architecture. The Model-Driven Development approach is presented in Section 3 and validated in the scope of a case-study example in Section 4. The evaluation and comparison with related work are covered in Section 5. Finally, Section 6 provides the conclusions of this work and points to our plans for future work.

2 | REFERENCE ARCHITECTURE

The study of software architectures[9] and software architecture patterns[10] has allowed software developers to conceive, design, implement and maintain highly complex software systems. The academic rigour of pioneers working in the wider area of *software architectures* is even credited as a main enabler of the mobile app revolution we are experiencing today, evident by the widespread success of the Android platform[11].

Often, software architectures are complemented with tools which facilitate their design and implementation, such as those defined in the *Model Driven Development* approach. An excellent introduction to MDD is provided by Mellor et al.[12] who argue that "*Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing*". In programming, many layers of models exist, whereas for example the abstract sketch on a piece of paper, or in our mind, is at higher level than a few lines of code, which is itself at a higher level of abstraction compared to the assembly language used on the virtual machine or CPU. In the context of this paper—and most software engineering papers—*models* relate primarily to higher-level drawings, typically expressed with UML, aiming to provide a platform-independent abstraction of a software system.

Model-Driven Development (MDD) has been proposed as a mechanism for easing the effort of developing software artifacts by means of a model-first approach[13]. Naturally, MDD-based approaches can be combined with component-based software development[14], where specific model artifacts correspond to concrete components. This raises both the level of abstraction from the developer's perspective, but also it maximizes software reuse in terms of easily incorporating ready-made components.

While the underlying ideas of this approach apply to general component-based architectures, in this paper we use an existing reference architecture to allow for a concrete demonstration of how these ideas are applied and evaluated. We argue that the proposed MDD approach is applicable to general component-based systems, such as OSGi which is widely used in embedded systems, and the Android platform which is the most widely used mobile framework.

The reference architecture is based on the MUSIC platform[15,16] which is itself a research-driven implementation of a modular, component-based architecture for building context-aware applications. The latter provides middleware, a software development methodology and tools for the design and implementation of self-adaptive, context-aware applications. As part of this comprehensive solution, a *Context System* is responsible for collecting and managing context data, as well as for providing the middleware—and the applications that are deployed on it—with access to relevant context data.

The context system itself builds on a pluggable architecture, similar to those proposed in pioneering works in context-awareness, such as the *Context Toolkit*[1], the *Context Fabric*[17] and the *Java Context Awareness Framework* (JCAF)[18]. In this type of architecture, individual tasks such as collecting certain context data types (e.g. battery level) or inferring higher level context from raw data (e.g. user activity) are mapped to individual plug-in components. As these components can be developed independently, and activated or deactivated separately, this architecture benefits both in terms of code portability[5] as well as in terms of better resource utilization[19].

In the following subsections we describe the *Context Model* and the *Context Query Language* used throughout this approach. Furthermore, we describe the structure of a typical context plug-in and briefly discuss the architecture of the system. These provide the foundation for the proposed MDD methodology and tool-chain.

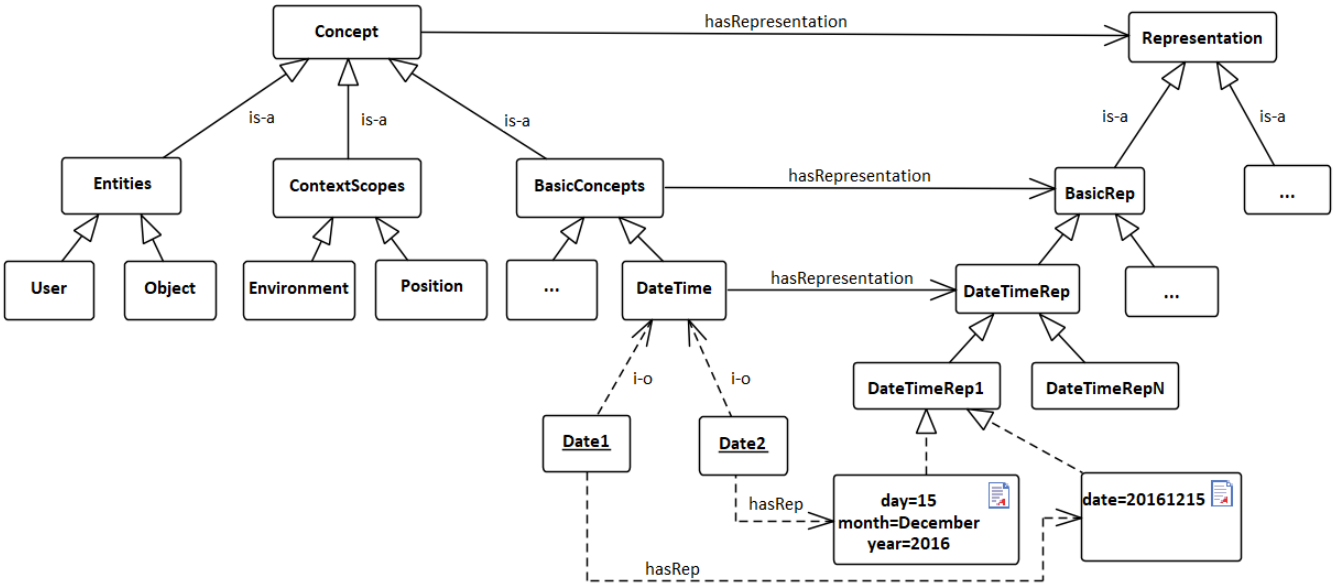

FIGURE 1 An example of the Context Ontology [20].

2.1 | The Context Model

The adopted context model is simple but, also, highly extensible. It is based on XML and makes use of ontologies that are described in OWL. The context information is represented in terms of context elements, which provide information of a certain type, called *context scope*, and which describe a certain context entity. The former describes a specific domain (of an entity), such as "position", "civil address", "environment", "user proximity", etc. The latter refers to concrete entities in the world such as for example a "user", a "room", a "device", etc. In our context model *Concepts*, as a generalization of *Scopes* and *Entities*, are associated with one or more *Representations* describing the structure of the context data in terms of *Parameters* (attributes of the context data and associated meta-data along with admitted types and values), *Structures of Parameters*, and *Arrays of* (structures of) *Parameters*. In order to establish a common understanding about the semantics of the different concepts in a heterogeneous pervasive environment, the Context Scopes, the types of Context Entities and the different Representations, they are all described using an ontology. An example illustrating the basic concepts and the main structure of the ontology is shown in Figure 1 (based on the example illustrated in [20]).

Like in the *Aspect-Scale-Context* (ASC) model by Strang et al [21], the ontology also includes so-called *Inter Representation Operations* (IROs), facilitating the automatic conversion of measure units, as well as more complex conversions between completely different representations. For example a "position" expressed in geocoordinates is converted to an "address" expressed in terms of street, city and country. In another example, a date is converted from a single-line representation to one which splits it to day, month and year (also illustrated in Figure 1). For this purpose, the ontology provides the grounding to a certain method in a library or to a certain service providing the appropriate functionality. In the same way, we allow the definition of *Aggregation Functions* in the ontology, in order to enable the aggregation of (sets of) context elements to a certain value or to derive more elaborate information. Additionally, the ontology is also used to describe relationships between entities, e.g. a child has a father and a mother, or a room belongs to a building. This allows the description and processing of semantically complex queries in a compact manner, as an ontology reasoner can be used to automatically resolve the relationships.

Ontology reasoning is an intensive process and thus it is not desirable, and often not even feasible, to perform on mobile devices at run-time. For this reason, this ontology allows characterizing context entities and context scopes simply via predefined types. The type implicitly corresponds to a certain context scope and to a default representation of the context information.

2.2 | The Context Query Language

While the proposed context system provides a comprehensive framework for context modelling, context management and context reasoning, it also includes an associated *Context Query Language* (CQL). Here, we give only a brief overview of this CQL and its relevant features, to the extent that is relevant to the MDD approach described in this paper. A thorough description of the query language is provided by Reichle et al. [22] and by Fra et al. [23].

Just as the context model itself, the corresponding CQL is XML-based and makes heavy use of ontologies. It allows querying information that characterizes one or several context entities and corresponds to a certain scope in a specified representation. In order to provide elaborate accessing and filtering methods, the CQL allows the specification of constraints on parameters and meta-data attributes, ranging from simple constraints to complex conditions. For instance, the latter incorporate complex aggregation functions and semantic relationships that are resolved by an ontology reasoner. CQL explicitly addresses heterogeneous representations of context information, supports the specification of complex filtering mechanisms and allows the incorporation of an extensible set of aggregation functions.

One of the main advantages of CQL is its strong weaving with ontologies. Using references to the ontology, semantic reasoning on context information is supported and heterogeneity is explicitly addressed. While existing context query languages support only a subset of all the features mentioned above, CQL stands out by supporting all of them. This is also supported by the evaluation presented in [24], which identified the CML [25] and the RDF-based MoGATU [26] as the two most sophisticated systems. However both of them lack either support for heterogeneous representations and incorporation of ontologies, or the provisioning of elaborate aggregation and filtering mechanisms.

The CQL also facilitates the request for context information in a specific representation and the incorporation of Inter-Representation Operations (IROs). The underlying query processing system automatically performs the necessary conversions between different representations. In Section 4.1.3 we briefly explain how the mediation task is supported by the MDD tool-chain via automatically generated converters.

Finally, an additional mechanism is provided for rudimentary queries that do not require any filtering of information, based on a basic Context Query Factory that generates the corresponding XML request automatically by simply providing the requested entity, scope and representation.

2.3 | The Pluggable Architecture

This paper builds on a modular, pluggable architecture. This allows for multiple context providers and context consumers to be dynamically bound to a central component, e.g. by realizing a publish-subscribe like pattern. The component responsible for this (Context Manager) routes the context information to the clients and also maintains a cache of recent values for further processing or retrieval. In this architecture, the context providers are materialized as *context plug-ins* which can be dynamically installed, deployed and activated automatically by the underlying middleware. On the other hand, *context clients* refer to the deployed, context-aware applications and the middleware modules which are responsible for the adaptation reasoning.

In [5] it was shown that this specific architecture has significant advantages as it enables the design of context-aware applications in a way in which their context-aware properties are specified independently of their functional logic. This *separation of concerns* eases both the development and maintenance efforts of the developers. Furthermore, the resulting context plug-ins are, to a large extend, reusable across devices and applications. More specifically, lower-level sensor plug-ins can be reused across similar devices of the same architecture, while higher-level sensors can be reused across multiple applications. Finally, a common underlying ontology, as described in [20], facilitates the reusability of the plug-ins also at a semantic level.

Concerning the runtime advantages of this approach, an evident benefit is the fact that when multiple applications require the same context types, then only one instance of the corresponding context plug-in needs to be activated (as opposed to applications that embed their separate copies of context logic). Another advantage is also the fact that depending on the actual context needs of the deployed applications, only the minimum set of required plug-ins is activated at any moment, which results to better resource utilization, an important asset for mobile, battery-powered devices.

The context plug-ins are self-contained components which export an interface to advertise their *required* and *provided* context types. Furthermore, each plug-in is associated to a set of (optional) metadata, used to describe properties such as their ID, manufacturer name, resource consumption, accuracy of measurements, etc. When registered with the Context Manager, the latter records their required and provided context types to a local map data structure. In parallel, deployed applications advertise their context needs, either explicitly via appropriate methods, or implicitly by parsing their utility functions. As not all the deployed applications are active all the time, the Context Manager dynamically evaluates the context needs of the running applications and activates (or deactivates) the appropriate context plug-ins accordingly. Naturally, this approach results to better resource utilization as it was experimentally shown in [19]. A more detailed example of context plug-ins and their use in the middleware is shown in the case study example in Section 4.

3 | MODEL DRIVEN DEVELOPMENT OF CONTEXT PLUG-INS

In model-driven development, newly defined domain specific modelling languages or extensions to existing languages are generally used to define models of the desired system at an abstract and platform-independent level. The defined *Platform-Independent Models* (PIMs) are automatically transformed to *Platform-Specific Models* (PSMs) and to platform-specific source code in one or more steps. For this purpose, a tool-chain is typically

incorporated into the corresponding development framework. Application developers are not confronted with implementation details and thus they can concentrate on the high-level view of the software to be developed. In this way, many conceptual and implementation errors can be avoided upfront.

The main approach involves the developer identifying the required functionality, coming up with an architecture (or several alternatives for it) and then picking the right components to model it. While in many cases the components might pre-exist, in some cases specialized components might be needed which are not available. In this case the developers have the option to build them based on the open specifications presented in this section. This is particularly relevant with *Operators* which are typically quite specialized (see Subsection 3.2).

Our intention is to exploit the benefits of MDD for the creation of context plug-ins, as they were introduced in Section 2. Starting from a purely conceptual model that introduces the main concepts and their relationships, we present a new UML Profile that can be utilized for the modelling of context plug-ins. Afterwards we describe the tool-chain and we briefly explain how the transformation is realized.

3.1 | Conceptual Model

The conceptual model is illustrated in Figure 2, which shows its constituent entities and their relationships. The main entities in this conceptual view are the *Context Plug-ins*, the *Operators*, the *Data Management Containers* (DMCs) and the *Connectors*.

A Context Plug-in represents the architectural element that is physically deployed on the context middleware. Externally it is mainly characterized by its provided and required context types. In order to interact with the context system, to express the required and the provided information and to have basic data structures for storing and caching context information, it defines an arbitrary number of Input DMCs and one Output DMC. In addition, it defines its trigger type, which can be time-triggered (OnTime) for continuous input, or event-based (OnChange) for discrete input. The trigger value concretizes the trigger type through an appropriate parameterization, i.e., it provides a certain trigger interval or concretizes the event that causes the triggering.

In the same way as in [27], a DMC typically represents a common data structure, like a simple variable, array, ring-buffer or queue. The DMCs are used for caching information but without major functionality. A DMC is characterized through its type (e.g., variable, array, etc.) its size (i.e., number of elements to be cached) and the *time-to-live* (TTL) for their contents. Additionally, DMCs comprise a "representation" attribute. This attribute refers to a certain semantic representation concept of the Ontology and thus defines the data-type of the stored elements. Input DMC and Output DMC are specializations of DMC, that are incorporated by a Context Plug-in to interact with the context system and to define its required and provided context information. An Output DMC characterizes the provided information by specifying the corresponding entity, the scope of the provided information and its representation (inherited from DMC). An Input DMC specifies the required context information and provides two further specializations: A SimpleInputDMC which is used to characterize the required information simply through the corresponding entity, the scope and the desired representation of the information. However, as our context system provides elaborate context access through the CQL (see previous section) we also allow expressing the required context information through an associated query. For this purpose, we have introduced the ComplexInputDMC. As a Context Query allows refining the requested information by specifying conditions on values and also on meta-data (as for example a timestamp) this also facilitates to deal with aspects like data freshness.

Operators are conceptual entities that provide a specific functionality and perform a specific calculation on the data cached in one or several DMCs. The results are stored in a DMC again. We distinguish between *Generic Operators* and *User Defined Operators*. Generic Operators are predefined (i.e., pre-implemented) operators that can be reused in many different applications and contexts. Usually they are provided as part of a library. A generic operator is associated with a number of configuration parameters that allows fine-tuning of its behaviour. Examples for generic operators are provided in Section 3.2. In contrast to generic operators, *User Defined Operators* define more specific functionalities that are usually useful only within a particular context plug-in. The corresponding function body can be specified in pseudo code, in special placeholders which are included in the automatically generated source code as comments.

While operators are the main computational entities and are used to define the internal functionality of a context plug-in, connectors are the main information channels, used to connect the corresponding DMCs to the operators. In this respect the connectors define the data-flow within a plug-in. Furthermore, connectors are also used to perform mediation tasks from the data stored in one DMC to the data required in another one, i.e. extracting particular scopes or dimensions from a data structure and/or performing *Inter Representation Operations* (i.e., transformations). The required mediation is specified through an associated string.

With regard to the data flow, it is assumed that the operators, in conjunction with the connectors, form a *Directed Acyclic Graph* (DAG). This allows us to model the operators and the connectors as a form of filter chain. This chain is eventually translated into a single '*compute*' method that deals with all the computation and mediation tasks, without having to cope with loops or other ambiguous effects that can only be hardly grasped at a purely conceptual layer. Further details on the use of connectors, operators, as well as on the overall conceptual model, are illustrated in Section 4 via an example.

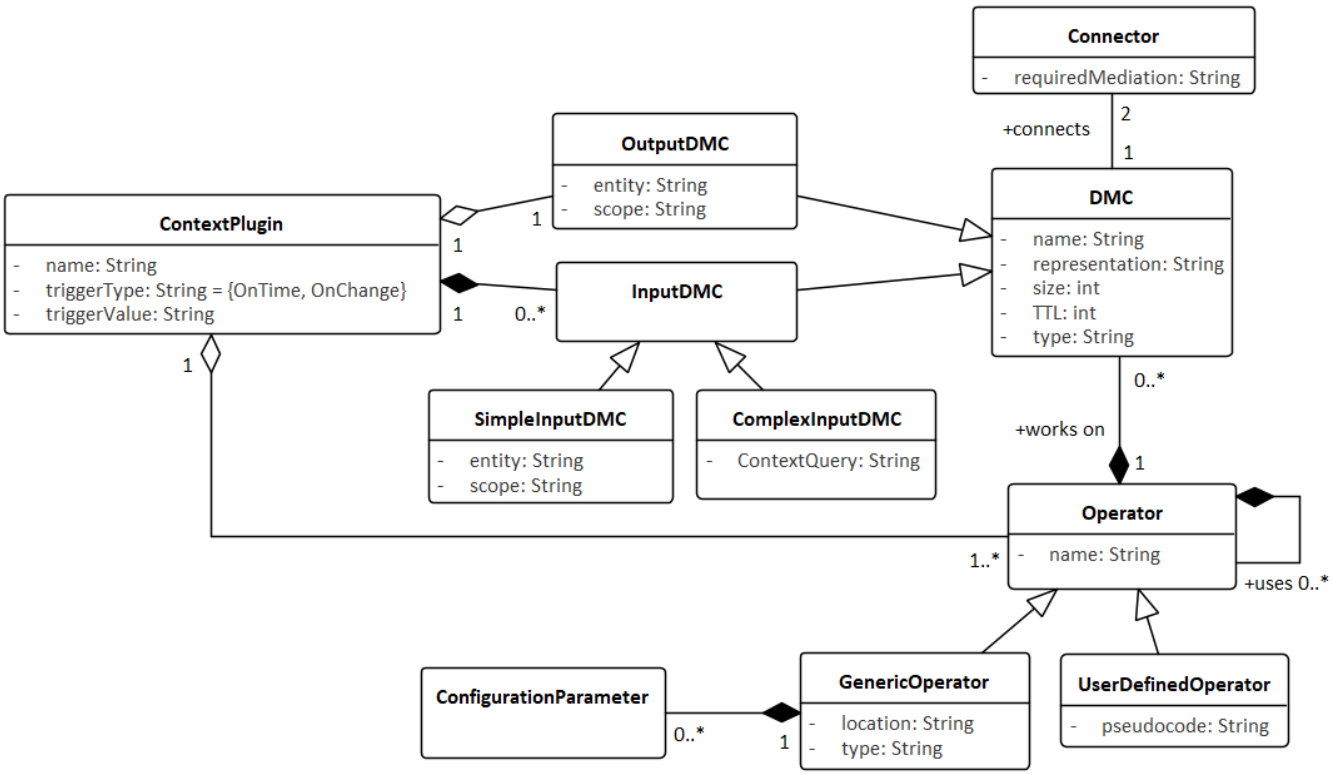

FIGURE 2 Conceptual Model of Context Plug-ins.

3.2 | Operators

At the core of context processing are specialized *operators*. Even though in this section we describe a few examples of *generic operators*, our approach is architected as an open platform which enables developers to specify and provide additional, reusable operators as per their needs. However, in order to provide a better understanding of how generic operators are formed and the functionality they might provide, we describe three specific examples:

Value Predictor operator

The first operator we examine is the *Value Predictor*. The basic functionality of this operator is to cache numerical values distributed over the time axis, and try to *predict* their value trend for the immediate future. The rationale for having such an operator is to try and predict trends in resources that might affect the operation of an application. For instance, by monitoring the signal strength of a wireless network, it is possible to predict events where the declining strength could hint to an upcoming network disconnection (e.g. walking away from an access point).

Based on the nature of the input data, specialized operators can be used to implement any of the following mathematical techniques: *Linear extrapolation*, *Polynomial extrapolation*, and *French-curve extrapolation*. The first is useful when the input data are described by a linear relation (i.e., lie on the same, straight line, such as the case of the remaining battery level in a mobile device). In our implementation, we have prototyped operators using the *linear* and the *polynomial* techniques.

Kalman Filter operator

A *Kalman Filter*[28] provides an efficient method to estimate the state of stochastic systems and can be considered as a special case of *Recursive Bayesian Filtering*. It is able to deal with noisy, missing and partly redundant measurements and minimizes the expected mean squared error. Kalman filters are often used to estimate the position and velocity of objects and thus provide the basis for many tracking systems.

For example, in the realm of context-aware systems Kalman Filters are often used for estimating the position of users from inaccurate readings of GPS sensors. In their more general form, they allow incorporating measurements from different sensors with varying accuracy.

We have developed a prototype of a *Kalman Filter Operator* which implements a linear Kalman Filter. For this purpose, it uses meta-data that express the covariance of the provided measurement dimensions. The operator is configurable to allow for estimating the state of static objects and of dynamic objects assuming a constant state change over time.

Image Comparator operator

With the purpose of allowing motion detection using a webcam, we provide an operator which compares two images and provides a *delta*, i.e., a (numerical) value indicating their difference. A trivial implementation of such an operator is quite straight-forward: the images are compared pixel-to-pixel, and for each of the three vectors (in the case of Red-Green-Blue, or RGB, encoding), we sum their difference. By dividing the delta sum by a fixed value (corresponding to the maximum delta possible), we are able to measure the difference of the images as a percentage.

Using an implementation of this operator, a *Motion Detector* reasoner plug-in was constructed using a queue to accumulate the two most recently captured images to be compared, and the *Image comparator operator*. The latter compares the two images and generates an event encoding their computed difference. This operator can be further customized by allowing a *threshold* property (i.e., generating an event only if the delta is above—or below—the specified threshold).

Pattern Matching operator

Patterns describe many natural phenomena, making it possible to automatically *learn* about them and then *classify* selected instances using mathematical or other models. This has given rise to many applications, from *optical character recognition* to *stock marker prediction*. These are generally classified as *Machine Learning*.

The *Pattern Matching* operator assumes that a training set already exists and then is used to facilitate automatic pattern matching. For instance, the training set could consist of a string (i.e. sequence of *symbols* as illustrated in Table 2) and the query could consist of a partial string (e.g. a sequence of the first few *symbols*). By using a pattern matching algorithm such as the one proposed by Karp et al.[29], it is possible for this *operator* to compute the nearest match as well as the accuracy of the matching (e.g. as a percentage).

3.3 | UML Profile

The conceptual model presented in Section 3.1 defines the main conceptual entities to be covered by appropriate modelling elements. However, before we proceed with their definition, we first elaborate on the selection of an appropriate modelling language. The main consideration is the choice between *General Purpose Modelling Languages* (such as for example UML or XML) and *Domain Specific Languages* (DSL). As UML already provides modelling support for defining the internals of components as parts that are connected through ports in composite structure diagrams (which is quite aligned to our objectives) we have chosen to reuse these concepts to the greatest extent possible, and to tailor the semantics of UML to our approach by defining a UML Profile. Thus, the main task is to map the conceptual entities to UML modelling elements and to define appropriate stereotypes along with tagged values.

Figure 3 shows the UML Profile corresponding to the conceptual model described in Section 3.1. For all the main conceptual entities, appropriate stereotypes are defined and their attributes are included as tagged values. In order to avoid repetition, in these paragraphs we just highlight the extension of UML meta-classes, to provide a clear understanding of how the conceptual entities are reflected by UML modelling elements. The stereotype `mContextPlugin` extends UML Class. This qualifies it to be used in a composite structure diagram for the definition of its internal mechanisms. Actually it would be sufficient if the meta-class UML Encapsulated Classifier would be extended, but this meta-class is not available for extension in many UML modelling tools. In the same way, the stereotype `mContextOperator` extends UML Class as well. Here too, UML Part would be sufficient for use in UML Composite Structures diagrams. The stereotype `mConfigurationParameter` extends the metaclass UML Attribute. This allows the modelling of the configuration parameters as attributes of the Operator class. For User Defined Operators we have introduced the stereotype `mPseudoCode` which extends UML Note. Thus, the pseudo code for an operator can be provided by simply associating a note to the operator class. As DMCs serve as a kind of interaction points between the operators, we model them as ports. Consequently, a DMC extends the UML Port. However, as we would like to provide a convenient method for modelling the tagged values, the stereotype also extends UML Class. For the connector we just use the standard UML connector. The attribute *requiredMediation* is reflected through the stereotype `mContextMediation` which extends UML Note. This allows the association of the required mediation to the connectors as notes.

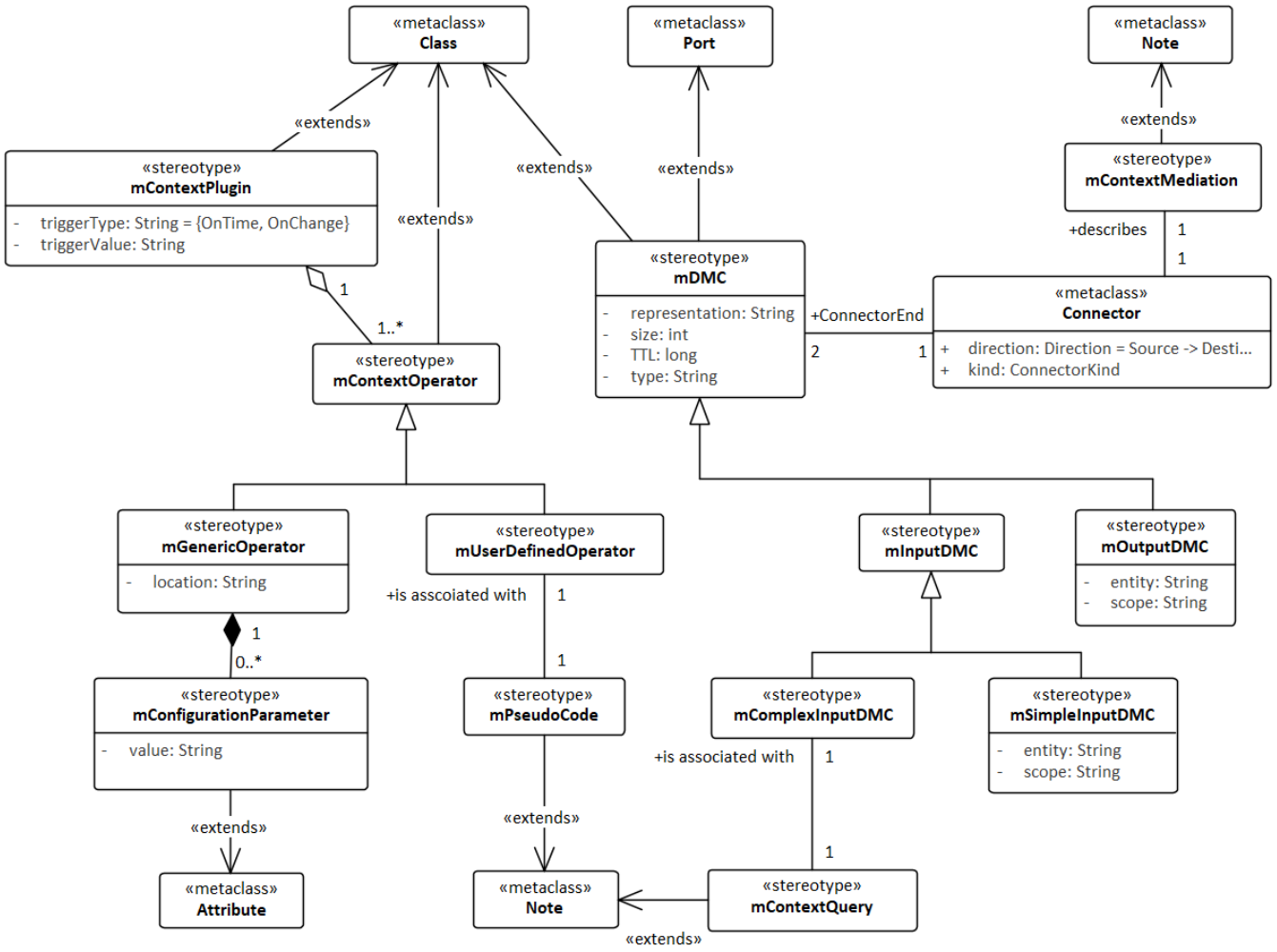

FIGURE 3 UML Profile for modelling Context Plug-ins.

3.4 | Tool-Chain and Transformation

The main constituents of our tool-chain are the UML modelling tool *Enterprise Architect* from Sparx Systems[1], and MOFScript[2]. Enterprise Architect supports OMG UML 2.x, but is not fully compliant to the Ecore interpretation of UML2, which is required as the input format for MOFScript. Thus, we have developed an XSLT Stylesheet that is able to convert Enterprise Architect exports to models compliant to Ecore UML2. Developing the MOFScript transformation script is mostly straight-forward (for more details see Section 4.1.3). However, a major challenge arises when, in addition to the UML models, we also have to incorporate information captured in ontologies, and represented in OWL, into the transformation process. For example, for the mediation tasks performed by connectors, it is necessary to have information about the internal structure of the context information and its available IROs. For this purpose, we use the Ecore meta-model for OWL 1.1[3], which enables MOFScript to process OWL ontologies. However, a prerequisite is that an appropriate OWL modelling tool supporting this standard is used. Here, we use the OWL modelling tool that serves as a standard implementation for the Ecore meta-model mentioned above. In the future, we plan to provide an appropriate XSLT Stylesheet that performs the transformation from, for example, Protégé exports to the Ecore OWL format. Both the base classes of the conceptual model and the transformation scripts are publicly available via Github[4].

[1] http://www.sparxsystems.com/products/ea
[2] https://marketplace.eclipse.org/content/mofscript-model-transformation-tool
[3] http://webont.org/owl/1.1/metamodel.html
[4] https://github.com/nearchos/music-mdd

4 | CASE STUDY-BASED EVALUATION

To illustrate the use of the MDD methodology and tool-chain we develop a research-based case study covering two scenarios. These concern a media player app that automatically pauses or resumes playback based on context, and another one that adjusts its media buffer size based on the user location. As the main contribution of this approach relates to the generation of context plug-ins, we focus the description on the context sensing and context reasoning aspects of the application rather than on the actual self-adaptive behavior of it. This is in line with the proposed design, which focuses on building plug-ins for context data collection and context inferring, leaving the final steps of acting on the context to the app developers.

The *Context-aware Media Player* (CaMP) has the following context-aware features:

- it detects when a user is present in her office and pauses (or resumes) the media playback accordingly and,
- it monitors the user movement inside a building and adjusts the streaming buffering strategy according to the prediction of network connectivity.

The former enables the media player to act *intelligently* and pause media playback when the user would not be able to listen to media, similarly to the scenario described in [30]. The latter optimizes online media playback on the move, by monitoring the user movement, and adjusting the streaming buffer size when the network quality changes—e.g. when the user is about to walk in a WiFi blind spot—similar to the scenario described in [31].

In the following, we split the discussion in two subsections, covering each of the main functionalities of CaMP listed above. Note that while these functionalities are operating in parallel, they are disjoint and thus they can be defined and developed independently, possibly on different devices. For the sake of demonstrating this variation, we assume that the former functionality is validated on a stationary computer (e.g. desktop), while the latter on a typical mobile device (e.g. a smart-phone).

4.1 | Scenario 1: Starting and stopping media playback.

The main components for enabling the start/stop functionality are the *main logic* (i.e., the control) component and the *media player* component. For the purpose of implementing this functionality, we also define four context plug-ins: the *User in the room detector*, the *Motion Detector*, the *Bluetooth device discovery* and the *Bluetooth device presence* plug-ins. These plug-ins and their dependencies are shown in Figure 4.

The *User in the room* plug-in is responsible for detecting whether the user is present in the room where the computer running the CaMP application is. In order to compute this information with reasonable accuracy, this plug-in uses input from two additional plug-ins, each one of which provides more elementary context information: The first, a *Motion Detector* plug-in, reports whenever significant movement is observed in the room by comparing consecutive images periodically captured by a webcam in the user's office. The second one, a *Bluetooth device discovery* plug-in, reports whenever a specific Bluetooth device (for example the Bluetooth-equipped smart-phone carried by the user) approaches her office (the other Bluetooth adapter is assumed to be attached to her desk computer).

While the *Bluetooth device discovery* plug-in indicates whether the user is near her office or not, it fails to detect whether the user is actually inside the office or just nearby (e.g., in an adjacent room as Bluetooth range typically extends to 5-10 meters). For this reason, the system utilizes the *Motion Detector* plug-in to check if there is also some activity (i.e., movement) in the office. Assuming that the user is the only occupant in the office, the *User in the room detector* plug-in can sense with high accuracy whenever the user enters or exits her office.

Finally, the *Bluetooth device discovery* plug-in utilizes the context information provided by another plug-in: the *Bluetooth device presence* plug-in. Instead of directly accessing the Bluetooth hardware, the *Bluetooth device presence* plug-in is deployed, which provides real-time context information of the detected Bluetooth devices (i.e., those that are within communication range). By registering for changes to the list of available Bluetooth devices, the *Bluetooth device discovery* plug-in can detect when a specific device (i.e., the user's smart-phone which is identified by its name or by its Bluetooth Address) becomes available or unavailable.

As soon as the user enters her office, the webcam detects motion and generates an appropriate event. At the same time, the Bluetooth adaptor on her computer detects the presence of the user by discovering her Bluetooth smart-phone. The combination of these events drives the *User in the room detector* plug-in to *infer* that the user is at her desk. This event is eventually communicated to the media player which then resumes playback.

4.1.1 | Manual, programmatic implementation of the Plug-ins.

In order to highlight the benefits of the MDD approach, we first introduce the main implementation aspects of the pluggable architecture with regard to the context plug-ins and hint at the steps which are required from a developer if a context plug-in were to be implemented manually.

In the reference architecture, plug-ins are defined as OSGi bundles which implement the `IContextPlugin` interface [15,16]. This is exported as a *declarative OSGi service* [32], and is automatically discovered and bound by the context manager when deployed.

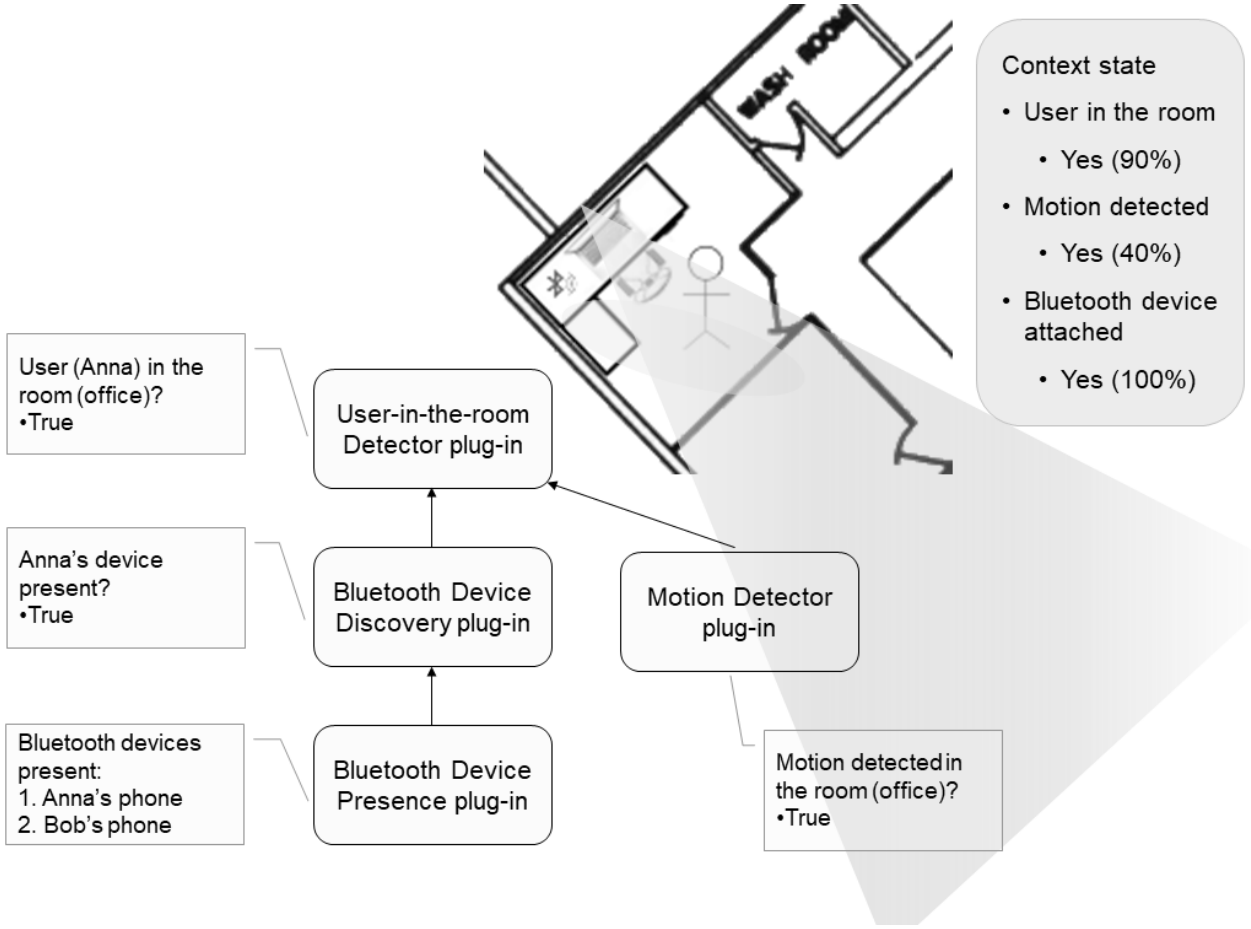

FIGURE 4 The plug-ins and their dependencies, used for automatically starting and stopping media playback.

The plug-ins' architecture defines methods for their activation and deactivation (which is dynamically invoked by the context manager using the *Inversion of Control* pattern). Furthermore, the reference architecture dictates that the plug-ins advertise their *required* and their *provided* context types [5]. For this purpose, each context plug-in is associated to an instance of the `IPluginMetadata` interface, which defines simple methods for accessing this metadata.

In the example of the *Motion Detector* plug-in (see Figure 5), the main component is implemented as an *extension* of the `AbstractContextPlugin` class, which itself implements the `IContextPlugin` interface. The *compute* method implements the main logic of the plug-in, and it is invoked by a thread which is triggered by the *activate* and *deactivate* methods (automatically called by the middleware as needed). When a context change is sensed, an event is constructed and delegated to the middleware using the `fireContextChangedEvent` method. Finally, the plug-in class is associated to an instantiation of the `IPluginMetadata` interface, which in this case simply states no required context types and a single provided one (i.e. *MOTION DETECTED*).

The typical process for creating the *Motion Detector* plug-in involves the engineer using the editor of his choice, and defining the two required classes by extending or implementing the middleware-defined abstract classes and interfaces. In more detail, a developer is confronted with the following tasks:

- Define the class that provides the meta-data of the context plug-in in terms of required and provided context types,
- Instantiation and parameterization of all the required data-structures (i.e., DMCs) and specialized methods (i.e., Operators),
- Establish the link to the context system through the `fireContextChange` and `contextChanged` methods, which are responsible to provide the results to the context system and to populate the DMCs accordingly,
- Implement the main logic of the plug-in (i.e., the *compute* method),

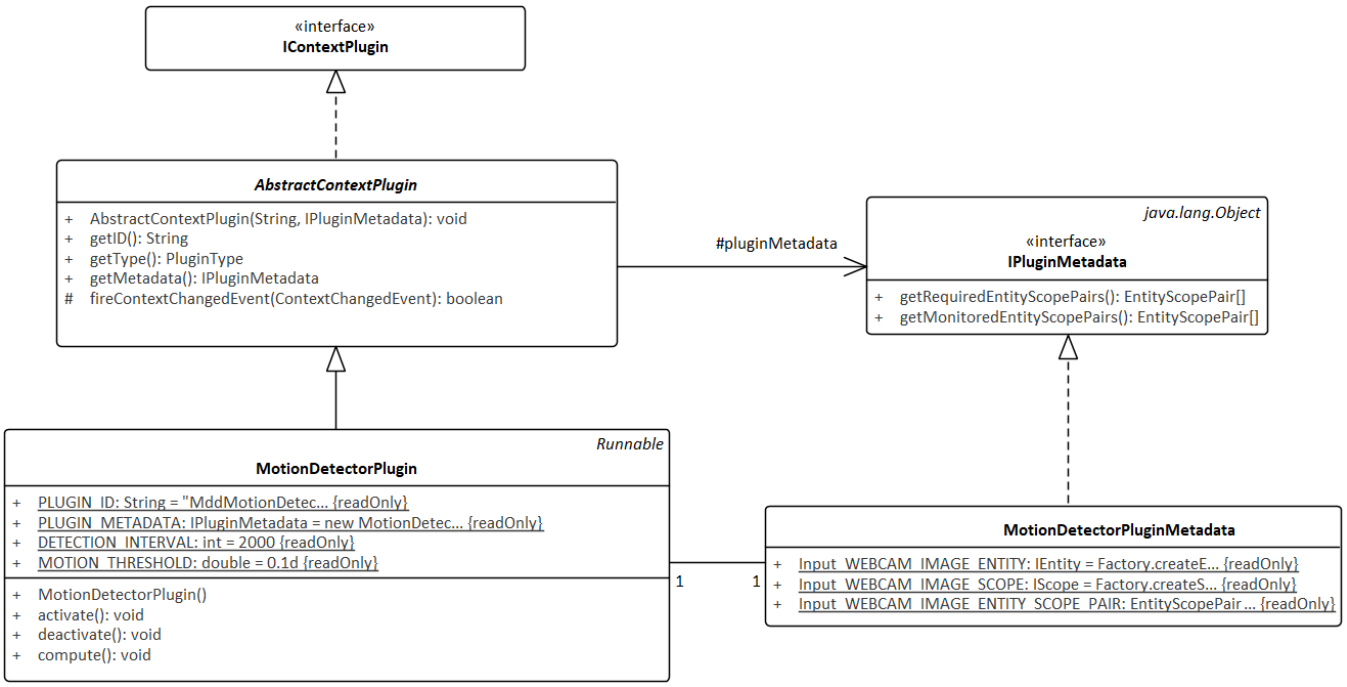

FIGURE 5 The basic architecture of the *Motion Detector* plug-in.

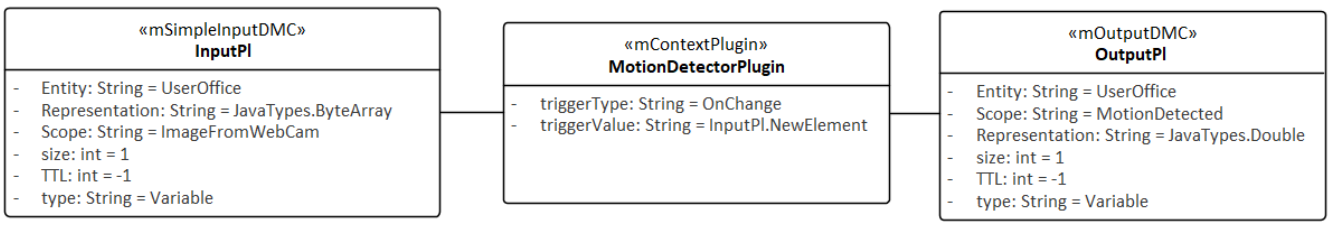

FIGURE 6 UML Class Diagram for the *Motion Detector* plug-in.

- Deal with the mediation required for the communication of the context information between the inputs and the outputs of the data structures (i.e., DMCs).

Although this particular functionality of the case study is not too complex itself, and even though abstract classes are provided to relieve the developers from recurring implementation tasks, this process can become a tedious and often difficult task when the desired functionality implies multiple data-structures and complex wiring code. Furthermore, once the code is compiled, the developer still needs to *package* the plug-in in an appropriate JAR file (assuming an OSGi-based architecture).

4.1.2 | Implementing the MotionDetector Plug-in Using MDD.

In this subsection we describe how the *Motion Detector* is modelled using the newly introduced UML Profile. This is viewed in comparison to the manual implementation of a context plug-in, as it was presented in the previous subsection.

First, the Context Plug-in itself and its associated Input DMCs and Output DMC are modelled. This is done in a UML Class Diagram as shown for the Motion-detector plug-in in Figure 6. The *Motion Detector* plug-in is modelled through a class with the stereotype `mContextPlugin`. The plug-in is parameterized by defining the trigger type: a change event in the Input DMC or, more precisely, if a new element is inserted into the Input DMC. In general, the *Motion Detector* Plug-in is associated with one SimpleInputDMC called *InputPl* and one OutputDMC named *OutputPl*. Both DMCs are modelled as classes with the corresponding stereotypes. The InputPl DMC is parameterized with just a simple variable that specifies unlimited time-to-live for its elements (i.e., the stored element will never be automatically invalidated). Additionally, we specify that *InputPl* requests context information characterizing the entity UserOffice. The scope of information is ImageFromWebcam and the requested representation is

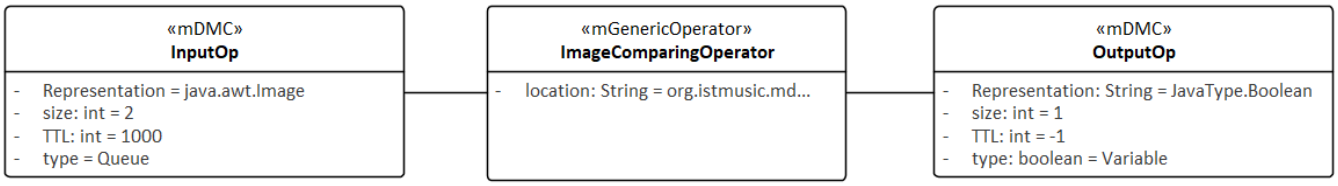

FIGURE 7 UML Class Diagram for the *Image Comparing* operator.

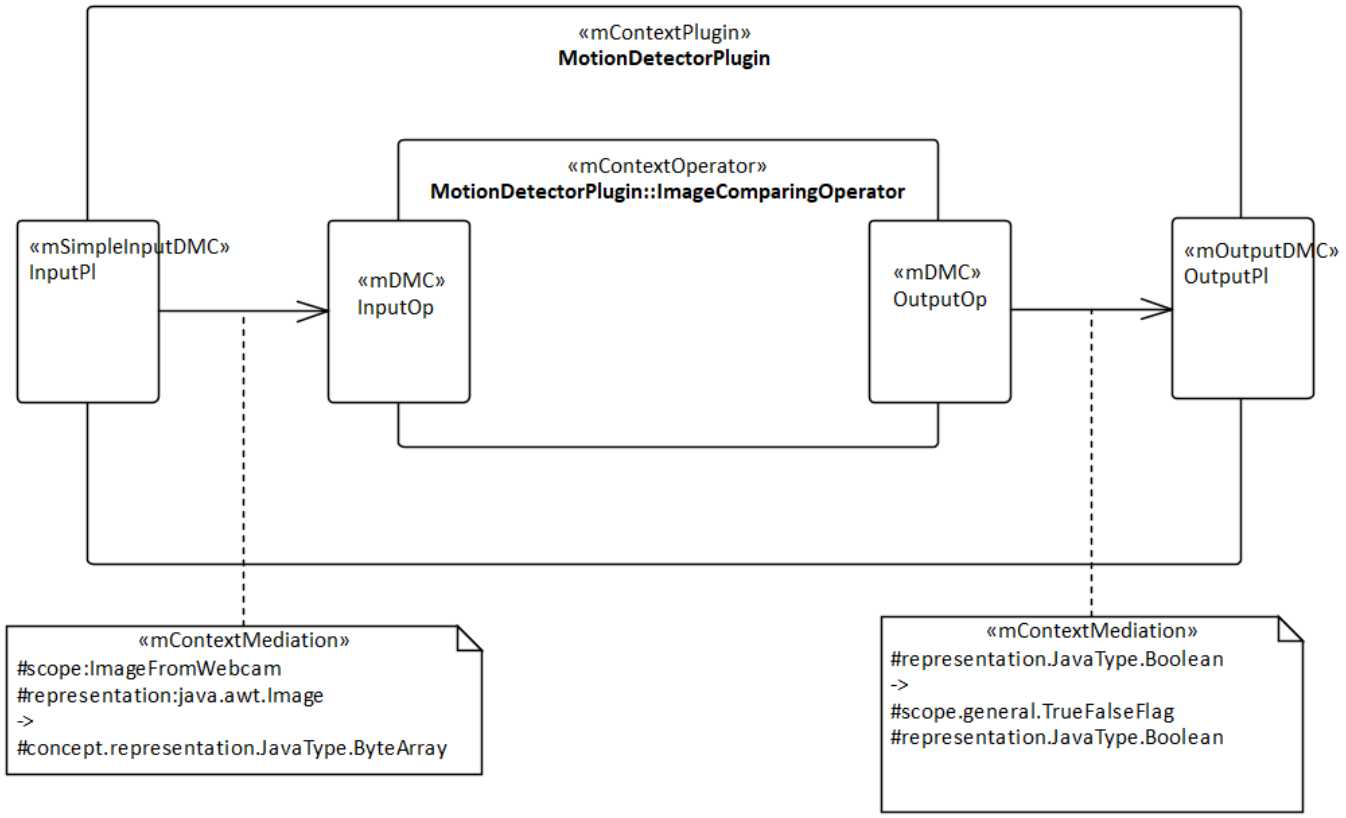

FIGURE 8 UML Composite Structure Diagram for Motion-detector Plug-in.

ImageFromWebcamDefaultRep. On the other hand, the OutputPl DMC is configured to specify that the information provided by the Context Plug-in to the Context Middleware also characterizes the entity UserOffice. Its scope is defined as MotionDetected and is represented according to the MotionDetectedDefaultRep.

Once the Context Plug-in is defined as described above, the developers need to model the Operators they would like to use within the Context Plug-in. This is also done with a UML Class Diagram. In the case of the *Motion Detector* plug-in just one operator is required, namely the generic ImageComparingOperator. Figure 7 shows the corresponding UML Class Diagram including its definition and parameterization.

The model for the operator is very similar to the model of the Context Plug-in itself. Therefore, here we only discuss their differences. In contrast to the Context Plug-in, an Operator cannot be triggered independently and, thus, the ImageComparingOperator does not specify the attributes *triggerType* and *triggerValue*. Instead, as a generic operator, the ImageComparingOperator provides the package location of the corresponding class to be instantiated. Another difference compared to Context Plug-ins is that an operator only works on generic DMCs that do not directly interact with the Context System. Hence, the DMCs only specify the representation (i.e., the data type) of the contained elements.

It is worth noting that as the ImageComparingOperator works on two consecutive images, the Input DMC is used to cache just two images. In this regard, this DMC is defined as a Queue of size 2 and its elements are invalidated (and removed) after 1000ms.

After the Context Plug-in and the operator have been specified, the next step is to model the data-flow between the plug-in and the operator. This is done by connecting the defined DMCs and by specifying the necessary mediation tasks. For this purpose, a UML Composite Structure Diagram is used, as shown in Figure 8.

In this diagram, the previously defined *ImageComparingOperator* is modelled as a nested classifier (represented as UML Class) of the *MotionDetectorPlug-in* class and the corresponding DMCs are connected through directed UML Connectors. In addition to the pure data flow, UML Notes associated to the connectors specify the needed mediation tasks. The note associated to the *Input-to-Input Connector* means that from the Context Element stored in the Input DMC of the *MotionDetectorPlugin*, the scope ImageBuffer in the representation BufferedImage has to be extracted and inserted into the Input DMC of the *ImageComparingOperator* in the same BufferedImage representation. Similarly, the mediation task is specified for the Output-to-Output connector.

4.1.3 | Transformation to Java code.

The final step of the MDD process is the generation of the source code. First, the UML model (see previous section) is exported to XMI and then converted to the XMI/UML2 representation expected as input by MOFScript. Next, the developer imports the resulting model into an Eclipse project. From there, the MOFScript with the provided transformation script is executed which results to the generation of the actual source code. Besides the source code, the transformation process also creates a directory structure which includes the Bundle manifest for the plug-in, along with its OSGI service declaration. Optionally, a build description is also provided which can be used directly by a build tool (such as Ant, Maven, etc.) to automatically generate the JAR-based bundle.

The transformation script which produces the plug-in's source code can logically be divided into two cycles: First, it reads all the elements available in the model and it stores them in appropriate containers, like lists, hashtables, etc. This is done to ease the subsequent code generation task.

Although the development of the transformation script was "straightforward", two problems had required significant elaboration: First, the correct transformation of the DAG for the generation of the *compute*-method of the context plug-in and second the automatic generation of source code to support the mediation between connected DMCs.

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

Concerning the second problem, a complete solution was not available at this point. In Section 3.4 it was mentioned that an ontology corresponding to the Ecore OWL format can be parsed by MOFScript. However, incorporating the concepts and relationships defined in the ontology into the automatic generation of mediation-methods has not been completed yet. Nevertheless, the current version of the transformation tool allows the generation of skeletons for every mediation-method, which is marked with additional "TODO" mark-ups and comments to indicate the points where the corresponding code has to be manually included.

Media players typically buffer a few seconds of the running audio or video clip to compensate for uncertainty in the streaming channel (e.g. when data is fetched from a locally attached device such as a DVD drive or from the Internet). The trade-off in this case is that buffering more data would require a larger portion of the available memory, leaving less space for the remaining apps. On the other hand, buffering less data means taking a higher risk to have an interruption to the media playback when there is a network disruption.

4.2 | Scenario 2: Adjusting media player's buffering strategy

An adaptive buffering strategy aims to optimize the perceived Quality of Experience (QoE) by adjusting the buffer size based on the network conditions. Many approaches focus on predicting exclusively based on the network properties (e.g. the adaptive media predictor discussed in [33]). In this scenario we envision a more elaborate mechanism to forecast network disconnection by predicting the user movement. To enable such functionality, it is assumed that a table exists with the expected network conditions at individual *Points of Interest* (POIs) (for instance separate rooms, entry points in buildings, etc. as depicted in Figure 9 and Table 1).

However, this still cannot predict the network conditions unless we also know where the user is heading. For this purpose, we propose a simple, machine learning-inspired approach where individual user motion patterns are recorded and then used to estimate the most likely path of the user. An example is illustrated in Figure 9: the user leaves their office (I) on the first floor, walks to the stairs (J, F, H) and then on the ground floor crosses the corridor (G, P, S) to get to a classroom (T).

POI	WiFi strength (*dB*)	Connection quality	Notes
F	-67	Very poor	Edge of stairwell (first floor)
G	-59	Poor	Edge of stairwell (ground floor)
H	$-\infty$	Disconnected	Stairwell (no signal)
I	-43	Very good	Office (frequent use)
J	-51	Good	Transition point
M	-64	Poor	Transition area
N	-53	Good	Admin office (frequent use)
R	$-\infty$	Disconnected	Stairwell (no signal)
S	-66	Very poor	Edge of stairwell
T	-47	Very good	Lab (frequent use)

TABLE 1 *Points of Interest* (POIs) with connection details and annotations (cf. [31])

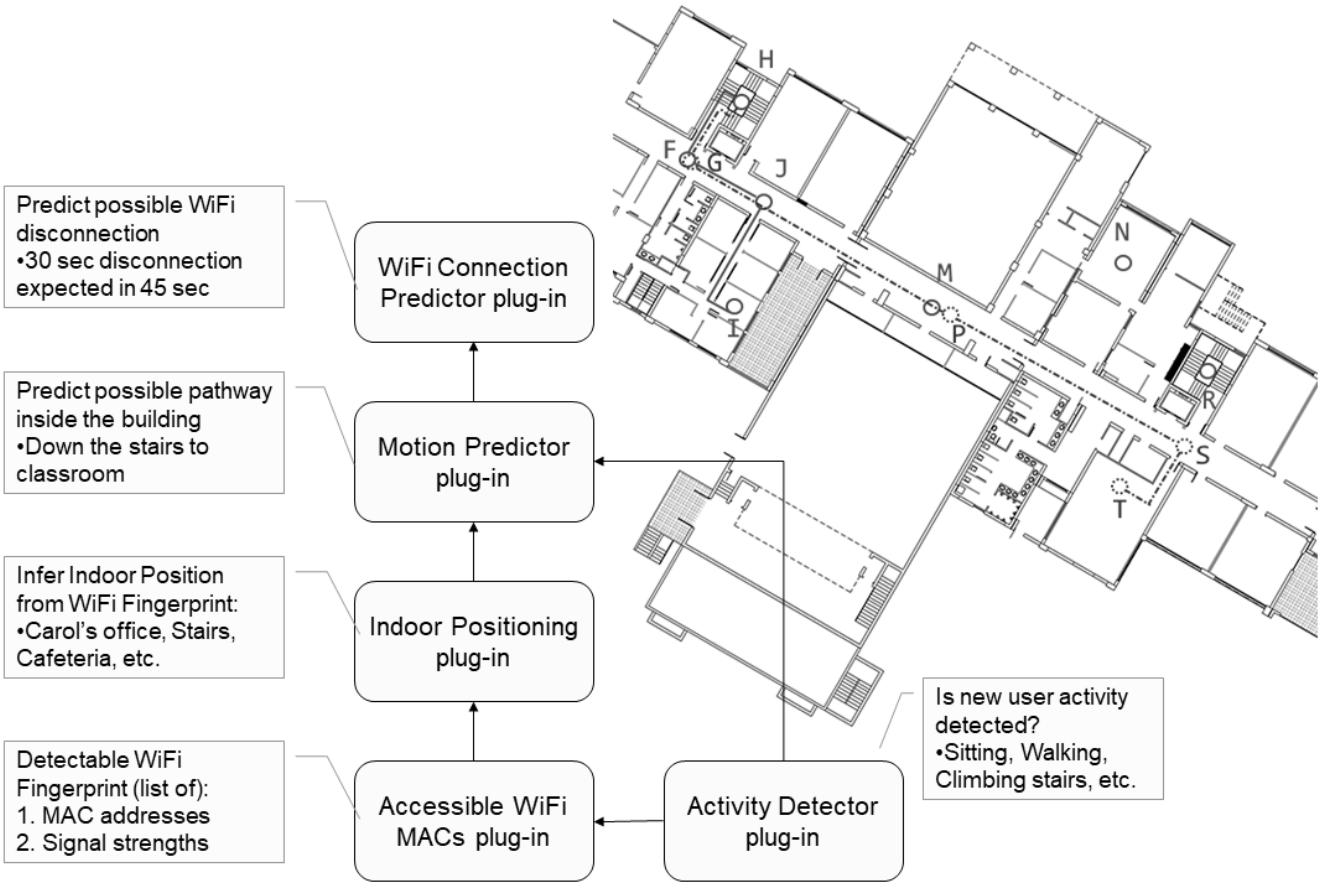

FIGURE 9 Illustration of the plug-ins and their dependencies, used for adjusting the streaming buffer size.

By using a time-stamped collection of the most likely motion patterns (e.g. as shown in Table 2) along with a prerecorded table of the expected network conditions at each POI, it is possible to make a prediction of the user position and subsequently the network connectivity. In its simplest form, network connectivity can be assumed to be proportional to the WiFi signal strength (e.g. as shown in Table 1).

In this scenario, prediction is facilitated by the fact that humans are creatures of habit, with predictable daily routines. For example, assume the data in this table cover a specific day of the week (e.g. Tuesday). When a user changes his status at around 9:00 that day from 'sitting' to 'walking' and then exits his office (I) and walks to the corridor (reaching POIs J and then F), then it can be predicted that he is on motion path (ii). This implies

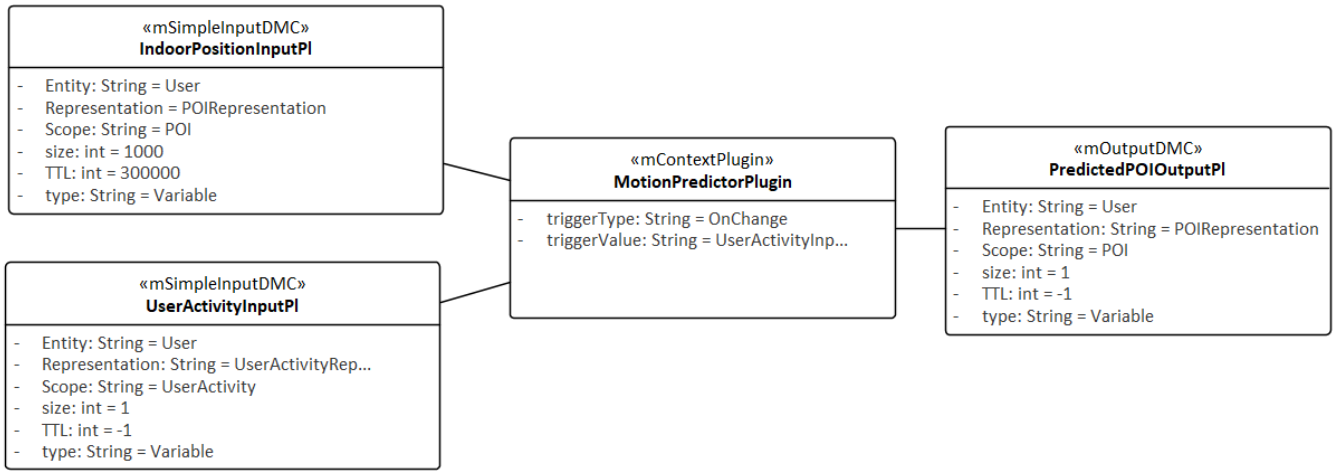

FIGURE 10 UML Class Diagram for *Motion Predictor* plug-in.

that the WiFi network is shortly going to be disturbed, i.e. when walking down the stairs (H). This event can subsequently trigger an adjustment to media player's buffering size. The exact algorithm used for prediction of Motion Path can vary from more elaborate, such as neural-network based AI approaches[34], to more "straightforward" algorithms such as pattern matching. In the following paragraphs, we build a plug-in that utilizes the pattern matching algorithm proposed by Karp et al.[29], and which was further described in the context of a similar scenario in[31].

Figure 9 depicts a stack of 5 plug-ins, working in tandem to predict when the WiFi network might be disconnected. At the lowest level, the *Accessible WiFi MACs* plug-in monitors the *visible* access points and triggers an event when there is a change (i.e. when a new access point becomes visible or an existing one disappears). These events are fed to the *Indoor Positioning* plug-in which uses a *fingerprinting* algorithm[35] to identify the most likely position of the user inside the building. Right above this, is a *Motion Predictor* plug-in which receives input from both the *Activity Detector* and the *Indoor Positioning* plug-ins. The former is used to trigger the plug-in into processing the (likely new) position of the user. Most modern mobile systems, such as Android, have built-in support for user activity tracking, so the *Activity Detector* plug-in could be realized as a wrapper around existing algorithms[36,37]. The latter provides the most recent position of the user. To fulfill the prediction task, the plug-in accumulates the trace of user position over time, so it can infer their *Motion path* and thus their likely future position. Lastly, generated events feed to the *WiFi Connection Predictor* plug-in which itself uses this information to make a prediction of the network quality, in this case by performing a simple look-up at the WiFi signal strength at the corresponding POI (Table 1).

Id	Time	Motion path	Notes
i	08:00	B C A D F J I	Arriving at the building
ii	09:00	I J F H G P S T	Going to a timetabled class
iii	12:00	T S R Q M J I	Returning at the office
iv	13:00	I J M Q R S V X	Going for lunch
v	13:30	X V S R Q M J I	Returning at the office
vi	14:30	I J M Q U W	Going to the library
vii	14:45	W U Q M J I	Returning at the office
viii	17:00	I J F D A C B	Leaving the building

TABLE 2 Fabricated model illustrating user's most common motion patterns in the building (cf.[31])

Undoubtedly, this scenario and especially the functionality of the *Indoor Positioning* and *Motion Predictor* plug-ins depend on a *training* phase which would allow them to build up the necessary database required for their operation (i.e. a list of WiFi access points and their WiFi strength visible at each POI for the former plug-in, and a list describing the Motion path of the user over time for the latter). For simplicity, we assume that such mechanisms are in place providing the *training* needed to realize this scenario, but for simplicity we do not describe them in detail.

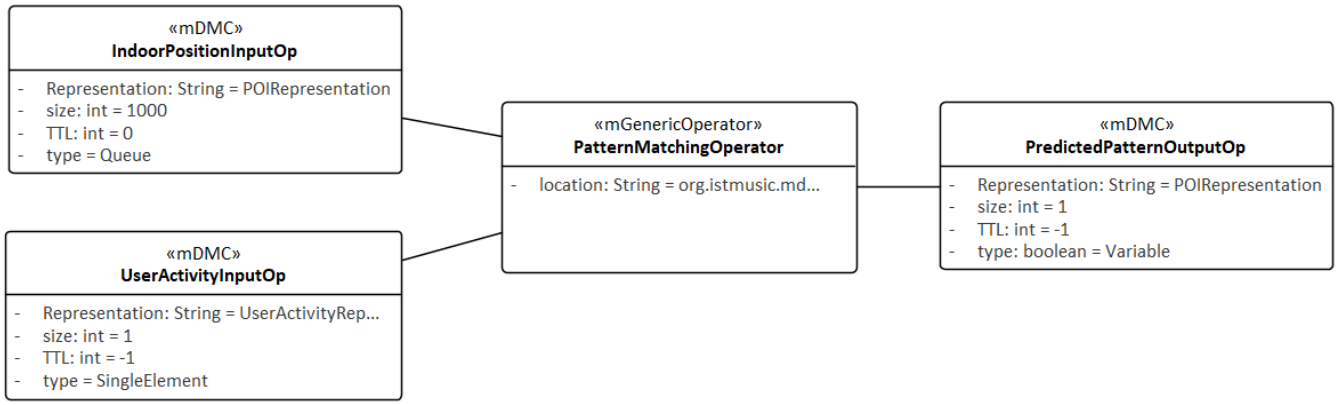

FIGURE 11 UML Class Diagram for *Pattern Matching* Operator.

4.2.1 | Implementing the Motion Predictor Plug-in Using MDD.

To further demonstrate the MDD approach, we describe the process that was followed to implement the *Motion Predictor* plug-in which utilizes input from the *Indoor Positioning* and the *Activity Detector* plug-ins (cf. Figure 9).

Similar to the first case study, the Context Plug-in itself and its associated Input DMCs and Output DMC are modelled with a UML Class Diagram as shown in Figure 10. Unlike the first case study though, this plug-in features input from two sources, and thus defines two Input DMCs. The *Motion Predictor* plug-in itself is modelled through a class with the stereotype `mContextPlugin`. The plug-in is parameterized by defining the trigger type: a change event in the *UserActivityInputPl* DMC or, more precisely, if a new element is inserted into this Input DMC. In general, the *Motion Predictor* Plug-in is associated with two SimpleInputDMCs called *IndoorPositionInputPl* and *UserActivityInputPl*, and one OutputDMC named *PredictedPoiOutputPl*. All DMCs are modelled as classes with the corresponding stereotypes. The UML Class Diagram for the *Motion Predictor* plug-in is depicted in Figure 10.

The model for the operator is similar to the model of the Context Plug-in itself. Therefore, here we only hint at their differences. As mentioned, in contrast to the Context Plug-in an Operator cannot be triggered independently and, thus, the PatternMatchingOperator does not specify the attributes *triggerType* and *triggerValue*. Instead, as a generic operator, the PatternMatchingOperator provides the package location of the corresponding class to be instantiated. As the PatternMatchingOperator works on several consecutive inputs (i.e. timestamped user positions), the *InputIndoorPositionPl* DMC is used to cache a sequence of events. In this regard, this DMC is defined as a Queue of unlimited size and its elements are invalidated (and removed) after 5 minutes (i.e. 300000ms), as it is assumed that distances inside the building are short (cf. Figure 11).

After the Context Plug-in and the operator have been specified, the next step is to model the data-flow between the plug-in and the operator. As before, this is done by connecting the defined DMCs and by specifying the necessary mediation tasks. For this purpose, a UML Composite Structure Diagram is used, as shown in Figure 12. In this diagram, the previously defined *PatternMatchingOperator* is modelled as a nested classifier (represented as UML Class) of the `MotionPredictorPlugin` class and the corresponding DMCs are connected through directed UML Connectors. In addition to the pure data flow, UML Notes associated to the connectors specify the needed *mediation* tasks, i.e. processes aimed at transforming data to a compatible format or measure (e.g. from Fahrenheit to Celsius). In this scenario, mediation tasks are “straightforward” as data passes from the Plug-in to the Operator with the same representation.

The transformation to Java in this scenario is similar to the process described in Section 4.1.3.

4.3 | Lessons from the Case Studies

Our experience showed that the MDD approach has some important advantages compared to the manual approach. Similar to general MDD approaches, the developers benefit from the ability to implement their software at a higher level, dealing primarily with abstract data-structures and operators. For the development of context plug-ins, this means that developers do not have to go into the source code to understand the interfaces to the context system and other details that are required for the manual implementation. Furthermore, it should be noted that this approach was developed as part of a more general development methodology that aims at creating context-aware, self-adaptive applications. This results in a highly coupled solution, where the developers are provided with a wide range of development tools which are similar in use and thus contribute to a smoother learning curve. Furthermore, the developers can streamline the development process by allowing the modelling and transformation of the applications to take place in parallel with the development of relevant context plug-ins. Finally, a significant advantage is that the proposed

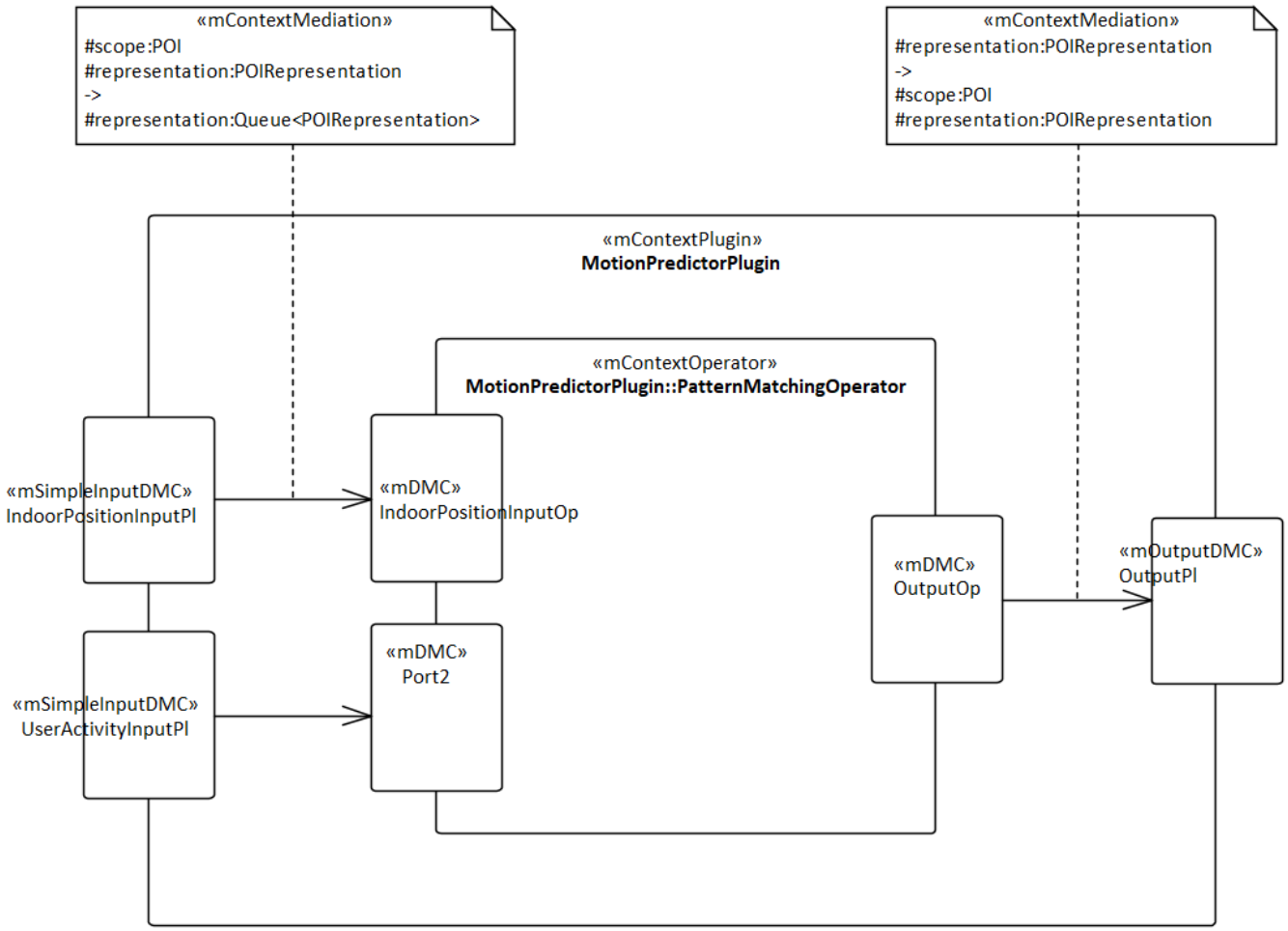

FIGURE 12 UML Composite Structure Diagram for *Motion Predictor* plug-in.

solution facilitates high re-usability of code, in terms of the main data-structures (such as the *Data Management Containers*) as well as in terms of specialized operators (such as the *Image Comparator* and the *Pattern Matching* operators).

In order to maximize reusability of the underlying artifacts—especially *Operators*—the developers should aim at balancing the need for *reusability* with that for *specialization*. For example, when developing a Motion Predictor operator, similar to the one described in Section 4.2, one could be tempted to aim for a general-purpose *Operator* which would work for instance similar to how *Deep Learning* [38] automatically infers data representations. On the other hand, another developer could aim for a highly specialized component which would work only in this particular case. Similar to standard component-oriented programming [14], finding the right balance is a difficult and at the same time an important task, as it allows to develop artifacts which are compact, meaningful and of course reusable.

5 | EVALUATION

The main contribution of the proposed approach is that it offers the ability to develop advanced context plug-ins—and consequently complex context-aware applications—in a systematic and automated way, while at the same time maximizing the opportunity for reuse. The value of this approach is evaluated both qualitatively and quantitatively, as well as via a comparison with related work and a discussion of its limitations.

5.1 | Qualitative Evaluation

To qualitatively assess the proposed approach we examine it across three dimensions: *expressiveness*, *reusability*, and *productivity*.

- *Expressiveness* is often measured by enumerating the number of possible scenarios or applications that can be developed using the aforementioned model [39]. In our approach, we argue that the proposed scenarios are very flexible and can be utilized to develop practically any conceivable application. This of course is partly due to the fact that the underlying model and the supporting framework are extensible, allowing for the specification and integration of third party components. Most notably, the developers can specify their own, custom-tailored operators which can be used to process the data in arbitrary ways. For instance, the *Operators* described in Section 3.2 cover a wide range of applications, from simple prediction, to image comparison, and signal smoothing. Naturally, this flexibility has the disadvantage of high cost of development, which is nevertheless counter-balanced by the re-usable nature of the constituent components, most notably the *Operators*.

- *Reusability* is one of the strongest aspects of this approach. Because the MDD-based approach is built around reusable components, such as the DMCs and the *Operators*, developers can take full advantage of the features of component-oriented programming and minimize development time, testing time, and their associated costs. For example, in case study 2, we proposed a sophisticated *Operator* which infers the user's movement pattern using string matching. In a similar manner, we could have created other *prediction* plug-ins such as battery level prediction, reusing to a great extend off-the-shelf *Operators*. The main disadvantage is that for the re-usability to be widely useful, a critical mass of tested, third-party operators must first be developed and made publicly available.

- *Productivity* is typically measured in terms of development time. A well known—and elaborate—approach for cost estimation was proposed by Boehm et al. [40] who argued that their *COCOMO II* approach "[…] *provides a thorough* […] *model to address modern software processes and construction techniques along with representative examples of applying the models to key software decision situations*". In some cases, a more "straightforward" approach is followed, where it is assumed that development time—and hence productivity—is proportional to the number of lines-of-code [41].

Last, and relevant to all three dimensions discussed above, *component-orientation* is another strong asset of the proposed approach. Proposed in the 90's [14], *Component Oriented Programming* is considered a proved technique for building modern software-based systems. For instance, Bagheri et al. [11] argue that Android owes its success to many software architectural principles developed over the previous decade, including component orientation–which is at the core of the proposed methodology.

5.2 | Quantitative Evaluation

The quantitative evaluation spans two approaches: First, we performed an experiment involving a small number of students, who compared the development of a plug-in using traditional programming versus using the proposed MDD approach. Second, we undertook the task of developing a context plug-in using both methods and then analyzed and compared the resulting code.

5.2.1 | Measuring Development Productivity

Similar to other works that evaluated software development approaches (e.g. Cassou et al. [39]), we applied a limited experiment in the context of a software engineering class at a university context. A small group of students (12) were initially asked to develop a number of context plug-ins using their existing programming skills. Then, a subset of them (4) were trained to use the MDD methodology and tool-chain and then they were asked to develop similar context plug-ins, this time using the provided MDD tool-chain. The participants then answered a short questionnaire where they were asked to compare the two approaches. The result showed that the MDD-based approach, despite its steeper learning curve, enabled them to complete the task faster and, furthermore, that the process could become even faster once they were further accustomed to using these tools. The complete format of the questionnaires and the student answers are openly available at [42]. The obvious limitation of this approach is the relatively small number of participants, and the fact that the academic context is often not an accurate reflection of *real world* software development conditions.

5.2.2 | Measuring Code Complexity and Performance

To further assess the effectiveness of the MDD tool-chain, we developed a context plug-in (specifically the *Motion Detector* plug-in discussed in Section 4.1) using both the proposed MDD-based approach, as well as traditional, manual development. We then compared the complexity and performance of the plug-ins to assess whether any of the two had an advantage over the other.

First, we used the models presented in Section 4.1.2 to generate the source code for the *MotionDetectorPlugin* which utilizes the *ImageComparingOperator*. Subsequently, we manually developed an equivalent plug-in (named *PlainMotionDetectorPlugin*) and tested both of them using a harness, under simulation conditions. Specifically, a simulated context sensor acts as the Webcam, *firing* events containing images from a predefined set. These are fed to the two plug-ins via the *harness*, which process them to identify the difference in the images and raise an event when

the difference exceeds a threshold. For simplicity, the *plain* plug-in reuses code from the *MotionPredictorPlugin*. Specifically, it uses the exact same implementation of the *IPluginMetadata* class (there is no need to manually develop it as it consists purely of boilerplate code defining the required and provided context types). It also reuses the function in the *ImageComparingOperator* which produces a numerical value comparing two images denoting their *difference*.

Plug-in	Lines-of-Code	Cyclomatic complexity	Memory use
	Source code only	*contextChanged() method*	*Average in Bytes*
Generated via MDD	108	4	1,362,054
Manually developed	74	6	1,360,620

TABLE 3 Comparison of automatically generated and manually developed *Motion Detector* plug-in

To compare the two plug-ins we used the more traditional metrics of *lines-of-code* and *cyclomatic complexity* [43]. For the former, we used IntelliJ IDEA's *Statistics* plug-in[5] and for the latter we used the *CyVis* tool[6]. We also performed an analysis of the memory use of the two implementations using the *VisualVM* tool[7].

The comparison showed that the MDD-generated code is larger in terms of *lines-of-code*. However, it should be noted that in this example the manually-created plug-in calls code directly from the *ImageComparingOperator*—had the corresponding code been copied-pasted, the lines of code of the plug-in would had increased from 74 to 132, i.e. to more than the MDD-generated plug-in. In terms of *cyclomatic complexity*, the MDD-generated code is actually fairing better, which is partly explained by the fact that a lot of the processing—e.g. handling and managing the flow of context events—happens in the DMCs in the case of the MDD-generated code, while the manually generated plug-in must handle the event flow explicitly. Finally, the memory usage is comparable. In reality, the plug-ins themselves use very small amounts of memory, but in this case the profiler included the memory used to queue the image buffers which makes the plug-ins appear as more memory demanding than would otherwise be the case. The code of the simulation harness and the plug-ins is available on Github[8].

5.3 | Related Work

From its early days when Context-Awareness was first introduced by Schilit [44] and later defined by Dey [45], it was clear that it would play an increasingly important role in mobile and pervasive computing applications. Since then, dozens of approaches have been proposed to collect, manage and infer context information. Some of these approaches are briefly described in [46] which also proposes a best-of-breed approach for building context-aware applications. An extensive survey of context modeling and reasoning techniques was conducted by Bettini et al. [6]. Similarly, Hong et al. reviewed and classified a large set of context-aware systems [7]. While many of these approaches were reportedly based on pluggable architectures, very few were designed to utilize MDD. One such approach was proposed by Ceri et al. [47], which however is constrained to Web applications.

At the same time, various forms of *Generative Programming* [48] have been explored, such as MDD and aspect-oriented programming, aiming primarily at improving the programmers' productivity and enabling code reuse. MDD was introduced with its proponents arguing that it "*distinguishes between conceptual models (where analysts work) and the code that implements the system (which can be generated with as much automation as possible from the conceptual model)*" [49]. Naturally, this is the primary benefit we are also claiming for the proposed plug-in based method presented in this paper.

In a position paper, Carton et al. [50] argued that combining aspect-oriented software development (AOSD) with model-driven development (MDD) has good potential (i.e. in terms of comprehensibility, maintainability and manageability metrics [51], which themselves have been found to be affecting software reusability [52]). We argue that the proposed approach also benefits from these as it features both an aspect-oriented approach (by means of the pluggable architecture [5]) and a model-driven development approach.

In another MDD approach [53], the authors propose an MDD-based approach for *Quality of Context* (QoC). While this approach has many similarities to our proposed one (especially in terms of modeling context information), at the same time it has some significant differences: it is largely

[5] https://plugins.jetbrains.com/plugin/4509-statistic
[6] https://sourceforge.net/projects/cyvis
[7] https://visualvm.github.io
[8] https://github.com/nearchos/music-mdd

based on a custom DSL rather than general-purpose UML, and it primarily aims at enabling QoC rather than general-purpose context-plugins (i.e. sensors/reasoners).

A similar, model-driven approach for developing context-aware applications was presented by Ayed et al.[54]. In that approach, the authors provided a detailed description of the steps required to generate the code of context-aware applications. These steps covered all the production phases. The authors conclude that the use of MDD in the development of context-aware applications allows platform independent development which "*hides the complexity and the heterogeneity of the context-aware and adaptive mechanisms*". Unlike our approach, this aims to enable the design of complete context-aware applications, and thus it puts emphasis on the *variability* aspects of the context-aware applications. In contrast, our approach focuses on the specification of the actual context plug-ins (i.e. context gathering and processing components) and their internal mechanisms which are used to collect and manipulate the context information. We argue that this degree of specialization provides a significant advantage as it allows the developers to maximize the portion of the functionality specified in the PIM and as a result minimize the amount of code required in the individual PSMs.

Also aiming directly at model-based plug-in development, Naujokat et al.[55] propose a method to enable a graphical modeling framework–namely jABC–to capture plug-in development in a domain-specific setting. The main contribution of this work is that "*the intended plug-in functionality can itself be modeled conveniently in terms of graphical workflows, which then can be translated fully automatically to running code*". While our approach builds on standard UML modeling tools such as *Enterprise Architect* and the more traditional transformation tool-chain *MOFScript*[56], the work by Naujokat et al. is based on their custom-tailored modeling framework *jABC*[57] and *Genesys* transformation tool[58,59].

In[60], Wagelaar and Jonckers use *Model Driven Architecture* (MDA) based approaches to design and deploy software applications targeting autonomous robots. MDA is used to produce a PIM of the software and then using a *Platform Model* (PM) they generate the required PSM. Similar to our approach, the use of a separate PM allowed the developers to reuse model transformations over several platforms.

In relevant approach described in[61], Geihs et al. use knowledge that is contained in ontologies to automate model transformations used in the model driven development of adaptive services and applications. For this purpose, the PIM of the service as well as the target platform are semantically annotated. This allows a generic transformation tool which incorporates simple ontology reasoner to target a number of platforms without adjusting the transformation itself. In contrast, our approach uses the principles of MDD and ontologies to build context-aware plug-ins, not necessarily targeting different platforms, but for increasing the productivity by reducing the amount of code required and by allowing to deal with the development at a higher level.

Finally, the authors of DiaSpec[39] propose a design language and a tool suite covering the development life-cycle of a pervasive computing application. Similar to ours, this approach is heavily based on a tool-set, but the focus is wider and covers the complete development cycle of a pervasive computing application. Our approach shares additional similarities in the form of aiming for expressiveness, re-usability and productivity. Unlike DiaSpec however, our approach is much more focused and specializes on the design and implementation of context plug-ins, which are individual components that alone are rarely sufficient to realize a full application.

5.4 | Limitations

Model-driven development has been a point of strong debate since its inception. An interesting discussion of the main arguments put forth and against MDD is provided by Mellor et al.[12]. Most criticism builds on the argument that MDD is more of a *hindrance* rather than *help*, as it adds extra tasks to the developer without an immediately observable benefit. On the other hand, proponents of MDD argue that "[MDD] *enables reuse at the domain level, increases quality as models are successively improved, reduces costs by using an automated process, and increases software solutions' longevity. In this way, models become assets instead of expenses–quite the business proposition!*"

Naturally, some of the limitations shared by virtually all MDD-based approaches are also observed in this proposal. For instance, a common criticism is that MDD carries a steep learning curve, it makes developers feel that they *don't have control*[62], and also that it is in contrast with the Agile-based approaches used widely in software development today[63]. As others have argued though, MDD is not without merit[13] and even not necessarily incompatible with Agile methods[64].

The case studies used for evaluation are based on existing demo applications, something that demonstrates the use of plug-ins for developing context-aware apps with separation of concerns[30]. While the case study scenarios are rather specific, they do demonstrate what state-of-the-art context-aware apps can achieve and showcase how the combination of simple plug-ins can form a more complex app, with relative ease. Another reason why the case study was chosen to be simple was so that it could be quickly demonstrated to students during training. This was important for the student-based evaluation which was carried out to evaluate the MDD based approach versus a "straightforward" (non MDD-based) approach—and described in Section 5.2.

Finally, it is argued that the proposed approach is confined by the rather narrow scope of the underlying middleware architecture, which while it is based on Java/OSGi and thus is highly portable, it is not very likely to be used in many modern apps. Nevertheless, we argue that the principle of the MDD methodology and tool-chain used in this approach is strong and solid. Also, we envision to port both the middleware and the tool-chain

to support at least one of the modern mobile platforms. Android is the natural candidate in this case, not only because it is Java-based and thus should be smoother to port to, but also because it is generally a more open platform [65].

6 | CONCLUSIONS

This paper presented a *Model Driven Development*-based approach for the creation of context plug-ins. At the core of this approach lies a method for creating complex context plug-ins, realizing sophisticated functionality, out of smaller, reusable artifacts via an MDD-based tool-chain. We showcased the approach in the context of two elaborate, research-based case-studies, demonstrating how complex functionality–such as automatically identifying when a user would become available for the media player to resume, or anticipating and reacting to WiFi disconnection–can be realized in a methodological and reusable manner. Furthermore, we evaluated the proposed tool-chain both qualitatively–via related work–and quantitatively–via a small student-based experiment and by analyzing and comparing plug-ins developed manually versus via MDD. This showed that this approach has significant advantages, primarily related to the code reuse which is at the center of the methodology.

In the future, we will extend and improve this methodology by implementing additional operators and DMCs and, also, will enhance the support for IROs. Additionally, we will evaluate the approach in the context of additional elements such as operators, and additional applications, including "standalone" context-aware applications. Finally, we will extend out tool-chain with support for additional *Platform Specific Models* to extensively and fully support Android as a target platform.

References

1. Dey Anind K., Abowd Gregory D., Salber Daniel. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction.* 2001;16(2):97–166.
2. Brunette Waylon, Sodt Rita, Chaudhri Rohit, et al. Open Data Kit Sensors: A Sensor Integration Framework for Android at the Application-level. In: MobiSys '12:351–364ACM; 2012; New York, NY, USA.
3. Carlson Darren, Schrader Andreas. A Wide-area Context-awareness Approach for Android. In: iiWAS '11:383–386ACM; 2011; New York, NY, USA.
4. Kim Hoon-Kyu, Kim Choung-Seok, Kim Kyung-Chang. A Context-Aware Framework for Mobile Computing Environment. In: LNEE:325-330Springer Singapore; 2015; Singapore.
5. Paspallis Nearchos, Papadopoulos George A.. A pluggable middleware architecture for developing context-aware mobile applications. *Personal and Ubiquitous Computing Journal.* 2013;:1-18.
6. Bettini Claudio, Brdiczka Oliver, Henricksen Karen, et al. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing.* 2010;6(2):161 - 180. Context Modelling, Reasoning and Management.
7. Hong Jong, Suh Eui, Kim Sung-Jin. Context-aware systems: A literature review and classification. *Expert Systems with Applications.* 2009;36(4):8509 - 8522.
8. Sommerville Ian. *Software engineering*. Pearson; 10th ed.2010.
9. Taylor Richard N., Medvidovic Nenad, Dashofy Eric M.. *Software Architecture: Foundations, Theory, and Practice*. Wiley; 1st ed.2009.
10. Richards Mark. *Software Architecture Patterns*. 1005 Gravenstein Highway North, Sebastopol, CA 95472, United States: O'Reilly Media; 1st ed.2015.
11. Bagheri Hamid, Garcia Joshua, Sadeghi Alireza, Malek Sam, Medvidovic Nenad. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software.* 2016;119:31 - 44.
12. Mellor Stephen J., Clark Anthony N., Futagami Takao. Model-driven development. *IEEE Software.* 2003;20(5):14-18.
13. Selic Bran. The pragmatics of model-driven development. *IEEE Software.* 2003;20(5):19-25.
14. Szyperski Clemens. *Component software: beyond object-oriented programming*. Addison-Wesley Professional; 1997.

15. Floch Jacqueline, Fra Cristina, Fricke Rolf, et al. Playing MUSIC - building context-aware and self-adaptive mobile applications. *Software: Practice and Experience Journal.* 2013;43(3):359-388.

16. Hallsteinsen Svein, Geihs Kurt, Paspallis Nearchos, et al. A Development Framework and Methodology for Self-Adapting Applications in Ubiquitous Computing Environments. *Journal of Systems and Software.* 2012;85(12):2840 – 2859.

17. Hong Jason I.. The Context Fabric: an infrastructure for context-aware computing. In: :554–555ACM; 2002; Minneapolis, Minnesota, USA.

18. Bardram Jakob E.. The Java Context Awareness Framework (JCAF) – a Service Infrastructure and Programming Framework for Context-aware Applications. In: PERVASIVE'05:98–115Springer-Verlag; 2005; Berlin, Heidelberg.

19. Paspallis Nearchos, Rouvoy Romain, Barone Paolo, Papadopoulos George A., Eliassen Frank, Mamelli Alessandro. A pluggable and reconfigurable architecture for a context-aware enabling middleware system. In: LNCS, vol. 5331: :553–570Springer Verlag; 2008; Monterrey, Mexico.

20. Reichle Roland, Wagner Michael, Khan Mohammad, et al. A Comprehensive Context Modeling Framework for Pervasive Computing Systems. In: LNCS, vol. 5053: :281–295Springer Verlag; 2008; Oslo, Norway.

21. Strang Thomas, Linnhoff-Popien Claudia, Frank Korbinian. CoOL: A Context Ontology Language to Enable Contextual Interoperability. In: LNCS, vol. 2893: :236–247Springer Verlag; 2003; Paris, France.

22. Reichle Roland, Wagner Michael, Khan Mohammad Ullah, et al. A Context Query Language for Pervasive Computing Environments. In: :434–440IEEE Computer Society; 2008; Hong Kong.

23. Fra Cristina, Valla Massimo, Paspallis Nearchos. High Level Context Query Processing: An Experience Report. In: :421-426IEEE Digital Library; 2011; Seattle, WA, USA.

24. Haghighi Pari Delir, Zaslavsky Arkady, Krishnaswamy Shonali. An Evaluation of Query Languages for Context-Aware Computing. In: :455–462; 2006.

25. Mcfadden Ted, Henricksen Karen, Indulska Jadwiga. Automating context-aware application development. In: :90–95; 2004; Nottingham, England, UK.

26. Perich Filip, Joshi Anupam, Yesha Yelena, Finin Tim. Collaborative joins in a pervasive computing environment. *The VLDB Journal.* 2005;14(2):182–196.

27. Baer Philipp A., Reichle Roland. Communication and Collaboration in Heterogeneous Teams of Soccer Robots. In: Vienna, Austria: I-Tech Education and Publishing 2007 (pp. 1–28).

28. Kalman Rudolph Emil. A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME – Journal of Basic Engineering.* 1960;82(Series D):35-45.

29. Karp Richard M., Miller Raymond E., Rosenberg Arnold L.. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In: STOC '72:125–136ACM; 1972; New York, NY, USA.

30. Paspallis Nearchos, Achilleos Achilleas, Kakousis Konstantinos, Papadopoulos George A.. Context-aware media player (CaMP): Developing context-aware applications with separation of concerns. In: :1684-1689; 2010.

31. Paspallis Nearchos, Alshaal Salah Eddin. Improving QoE via Context Prediction: A Case Study of Using WiFi Radiomaps to Predict Network Disconnection. In: ICPE '17 Companion:31–34ACM; 2017; New York, NY, USA.

32. Cervantes Humberto, Hall Richard S.. Autonomous adaptation to dynamic availability using a service-oriented component model. In: :614–623IEEE Computer Society; 2004; Edinburg, Scotland, UK.

33. DeLeon Phillip, Sreenan Cormac J.. An adaptive predictor for media playout buffering. In: :3097-3100 vol.6; 1999.

34. Russell Stuart J., Norvig Peter. *Artificial Intelligence: A Modern Approach*. Pearson Education; 3rd ed.2009.

35. Varshavsky Alexander, Patel Shwetak. Location in Ubiquitous Computing. In: Krumm John, ed. *Ubiquitous Computing Fundamentals*, Boca Raton, FL, USA: CRC Press 2009 (pp. 285-320).

36. Kwapisz Jennifer R., Weiss Gary M., Moore Samuel A.. Activity Recognition Using Cell Phone Accelerometers. *SIGKDD Explor. Newsl.*. 2011;12(2):74–82.

37. Lara Oscar D., Labrador Miguel A.. A Survey on Human Activity Recognition using Wearable Sensors. *IEEE Communications Surveys Tutorials*. 2013;15(3):1192-1209.

38. LeCun Yann, Bengio Yoshua, Hinton Geoffrey. Deep Learning. *Nature*. 2015;521(7553):436-444.

39. Cassou Damien, Bruneau Julien, Consel Charles, Balland Emilie. Toward a Tool-Based Development Methodology for Pervasive Computing Applications. *Transactions on Software Engineering*. 2012;38(6):1445-1463.

40. Boehm Barry W., Abts Chris, Brown A. Winsor, et al. *Software Cost Estimation with Cocomo II with Cdrom*. Upper Saddle River, NJ, USA: Prentice Hall PTR; 1st ed.2000.

41. Kieburtz Richard B., McKinney Laura, Bell Jeffrey M., et al. A Software Engineering Experiment in Software Component Generation. In: ICSE '96:542–552IEEE Computer Society; 1996; Washington, DC, USA.

42. Paspallis Nearchos. Summary of the classroom-based survey: questions and answers http://nearchos.github.io/mdd_survey.htmlLast accessed: 25 Oct 2018; .

43. McCabe Thomas J.. A Complexity Measure. *IEEE Transactions on Software Engineering*. 1976;SE-2(4):308–320.

44. Schilit Bill N., Adams Norman I., Want Roy. Context-aware computing applications. In: :85–90IEEE Computer Society; 1994; Santa Cruz, CA.

45. Dey Anind K.. Understanding and Using Context. *Personal Ubiquitous Computing*. 2001;5(1):4–7.

46. Ferreira Denzil, Kostakos Vassilis, Dey Anind K.. AWARE: mobile context instrumentation framework. *Frontiers in ICT*. 2015;2(6).

47. Ceri Stefano, Daniel Florian, Facca Federico M., Matera Maristella. Model-driven Engineering of Active Context-awareness. *World Wide Web*. 2007;10(4):387–413.

48. Krysztof Czarnecki Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: Addison-Wesley Professional; 1st ed.2000.

49. Panach Jose Ignacio, Juristo Natalia, Valverde Francisco, Pastor . A framework to identify primitives that represent usability within Model-Driven Development methods. *Information and Software Technology*. 2015;58:338 - 354.

50. Carton Andrew, Clarke Siobhan, Senart Aline, Cahill Vinny. Aspect-Oriented Model-Driven Development for Mobile Context-Aware Computing. In: SEPCASE '07:5-8IEEE Computer Society; 2007; Washington, DC, USA.

51. Munnelly Jennifer, Fritsch Serena, Clarke Siobhan. An Aspect-Oriented Approach to the Modularisation of Context. In: :114-124; 2007.

52. Bombonatti Denise, Goulão Miguel, Moreira Ana. Synergies and tradeoffs in software reuse - a systematic mapping study. *Software: Practice and Experience*. 2016;:943–957.

53. Hoyos José R., García-Molina Jesús, Botía Juan A., Preuveneers Davy. A model-driven approach for quality of context in pervasive systems. *Computers & Electrical Engineering*. 2016;55(Supplement C):39–58.

54. Ayed Dhouha, Delanote Didier, Berbers Yolande. MDD approach for the development of context-aware applications. In: LNCS, vol. 4635: :15–28Springer Verlag; 2007; Roskilde University, Denmark.

55. Naujokat Stefan, Neubauer Johannes, Lamprecht Anna-Lena, Steffen Bernhard, JÃűrges Sven, Margaria Tiziana. Simplicity-first model-based plug-in development. *Software: Practice and Experience*. 2014;44(3):277–297.

56. Oldevik Jon, Neple Tor, Grønmo Roy, Aagedal Jan, Berre Arne-J.. Toward Standardised Model to Text Transformations. In: Hartman Alan, Kreische David, eds. *Model Driven Architecture – Foundations and Applications*, :239–253Springer Berlin Heidelberg; 2005; Berlin, Heidelberg.

57. Naujokat Stefan, Lamprecht Anna-Lena, Steffen Bernhard, Jörges Sven, Margaria Tiziana. Simplicity principles for plug-in development: The jABC approach. In: :7–12; 2012.

58. Jörges Sven. *Construction and Evolution of Code Generators*. Berlin Heidelberg: Springer-Verlag; 1st ed.2013.

59. Jörges Sven, Margaria Tiziana, Steffen Bernhard. Genesys: service-oriented construction of property conform code generators. *Innovations in Systems and Software Engineering*. 2008;4(4):361–384.

60. Wagelaar Dennis, Jonckers Viviane. Explicit Platform Models for MDA. In: LNCS, vol. 3713: :367–381Springer Verlag; 2005; Genova, Italy.

61. Geihs Kurt, Baer Philipp, Reichle Roland, Wollenhaupt Jens. Ontology-based automatic model transformations. In: :387–391IEEE Computer Society; 2008; Cape Town, South Africa.

62. Martínez Yulkeidi, Cachero Cristina, Meliá Santiago. MDD vs. traditional software development: A practitioner's subjective perspective. *Information and Software Technology*. 2013;55(2):189 - 200. Special Section: Component-Based Software Engineering (CBSE), 2011.

63. Martin Robert Cecil. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR; 2003.

64. Ambler S. W.. Agile model driven development is good enough. *IEEE Software*. 2003;20(5):71-73.

65. Anvaari Mohsen, Jansen Slinger. Evaluating Architectural Openness in Mobile Software Platforms. In: ECSA '10:85–92ACM; 2010; New York, NY, USA.

How to cite this article: This is the pre-peer reviewed version of Paspallis N. (2018), An MDD-based approach for building context-aware applications with high reusability, to appear in the *J. Software: Evolution and Process*.