The Journal of Systems and Software 149 (2019) 285–304

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Evaluating the extension mechanisms of the knowledge discovery metamodel for aspect-oriented modernizations

Bruno M. Santos[a,*], André de S. Landi[b], Daniel S. Santibáñez[a], Rafael S. Durelli[c], Valter V. de Camargo[a]

[a] Federal University of São Carlos, São Carlos, SP, Brazil
[b] S2IT SOLUTIONS CONSULTORIA LTDA, Araraquara, SP, Brazil
[c] Federal University of Lavras, Lavras, MG, Brazil

ARTICLE INFO

Article history:
Received 15 April 2018
Revised 18 September 2018
Accepted 12 December 2018
Available online 12 December 2018

Keywords:
Aspect-oriented modernization
Knowledge discovery metamodel
Legacy systems
Heavyweight extension
Lightweight extension
OMG

ABSTRACT

Crosscutting concerns are an intrinsic problem of legacy systems, hindering their maintenance and evolution. A possible solution is to modernize these systems employing aspect-orientation, which provides suitable abstractions for modularizing these kind of concerns. Architecture-Driven Modernization is a more specific kind of software reengineering focused on employing standard metamodels along the whole process, promoting interoperability and reusability across different tools/vendors. Its main metamodel is the Knowledge Discovery Metamodel (KDM), which is able to represent a significant amount of system details. However, up to this moment, there is no extension of this metamodel for aspect-orientation, preventing software engineers from conducting Aspect-Oriented Modernizations. Therefore, in this paper we present our experience on creating a heavyweight and a lightweight extension of KDM for aspect-orientation. We conducted two evaluations. The first one showed all aspect-oriented concepts were represented in both extensions. The second one was a experiment, in which we have analyzed the productivity of software engineers using both extensions. The results showed that the heavyweight extension propitiate a more productive environment in terms of time and number of errors when compared to the lightweight one.

1. Introduction

For software systems to keep meeting the requirements previously established it is necessary constant evolution or they will no longer fulfill their role properly (Lehman, 1996). Many organizations have systems that, despite presenting the phenomena of erosion and aging, still provide significant value for the organizations. These systems are usually referred to "legacy systems". The erosion and aging consists in a system's detrition in consequence of successive and bad managed modifications in the source-code (Visaggio, 2001; Bianchi et al., 2003; Pérez-Castillo and Piattini, 2011).

For some organizations, the complete substitution of their system has a high risk and consumes a large amount of resources, making this alternative unfeasible. On the other hand, system reengineering is an alternative that is able to extend the system's life cycle and it is more feasible economically (Pérez-Castillo et al., 2011).

However, traditional reengineering processes lacks formalization and standardization on how to develop tools and how to make them work together, leading software engineers to create their own proprietary solutions, which are difficult (or even impossible) to be reused, hindering the productivity of the team (Pérez-Castillo et al., 2011).

In 2003, the Object Management Group (OMG)[1] created a task force to evolve the traditional reengineering processes, formalizing and preparing them to be supported by models (OMG, 2009, 2016). Therefore the term Architecture-Driven Modernization (ADM) came out as a solution to the standardization problem (OMG, 2009, 2016).

Architecture-Driven Modernization advocates modernization processes must employ MDA (Model-Driven Architecture) concepts along the process: Platform-Specific Model (PSM), Platform-

* Corresponding author at: Computer Department, Rodovia Washington Luis, São Carlos, 13565-905 SP, Brazil.
E-mail addresses: bruno.santos@ufscar.br (B.M. Santos), andre.landi@s2it.com.br (A.d.S. Landi), daniel.santibanez@ufscar.br (D.S. Santibáñez), rafael.durelli@dcc.ufla.br (R.S. Durelli), valtervcamargo@ufscar.br (V.V. de Camargo).

https://doi.org/10.1016/j.jss.2018.12.011
0164-1212/

[1] OMG is an international organization that approves open standards to object oriented applications since 1989.

Independent Model (PIM) and Computational-Independent Model (CIM). The goal is to rise the abstraction level to work in a technology-independent manner. Thus, the main idea is to represent the system to be modernized in models and conduct analysis and transformations on these models (Pérez-Castillo et al., 2011).

The Knowledge Discovery Metamodel (KDM) is the main ADM metamodel and its goal is to represent all systems characteristics, from low level details (like lines of code and programming structures) to higher level concepts (like architectural modules and business processes). In fact, KDM can be seen as a multimodel since it incorporates other metamodels and each one is responsible for representing a different system view.

Originally (and purposely) KDM does not include metaclasses for specifying particular domains or technologies, such as web services, multi-agent systems and aspect-oriented programming (AOP). However, it can be adapted in two different ways. The first one is by extending it in a lightweight (LW) manner by using stereotypes and tag values. The second one is by extending it in a heavyweight (HW) manner by changing the metamodel creating new metaclasses and/or modifying the existing ones.

When a legacy system presents modularization problems, usually due to the presence of crosscutting concerns, a candidate technology to be used in the modernization process is aspect-oriented software development (AOSD). AOSD is a relevant development methodology that has a significant impact on the community research and it also has a great number of publications around the world (Kulesza et al., 2013). There are also publications that reports on real usage of AOSD in industrial projects (Lesiecki, 2006; Hohenstein and Jäger, 2009). Important frameworks such as Spring and JBoss utilize aspect-oriented concepts, for example, a typical application might have a security policy that prevents a user from executing a number of operations unless the user has the correct privileges.

Even though the current KDM was devised to be a common intermediate representation for existing software systems its current version does not support the specification and instantiation of aspect-oriented concepts during modernization processes (Durelli et al., 2014b). Nowadays, KDM neither contain specific metaclasses nor stereotypes to fully support and represent aspect-oriented concepts such as: join points, advices, aspects, etc.

Moreover, we observe lack of studies in literature about: (*i*) representation of AOP in KDM (Shahshahani, 2011) and (*ii*) comparisons between different extension mechanisms of KDM. Regarding the first point, this lacking makes aspect-oriented modernizations an error-prone activity. This happens because the absence of specific metaclasses for representing aspect-oriented concepts needs to be compensated by representing the same aspect-oriented concepts using canonical metaclasses and trying to differentiating them somehow. This clearly can lead to misunderstandings and the insertion of errors.

In order to overcome these limitations in this paper we proposed two KDM extension for AOP – a lightweight and a heavyweight. By using these extension, the modernization into object-oriented systems to aspect-oriented ones becomes feasible, since it is possible to represent the aspect-oriented concepts (join points, advices, aspects and others) in a clear way in the KDM instance that represents the aspect-oriented version of the system. Another goal is to investigate both extensions, showing evidences of their suitability. To support this goal, a comparative study was performed to list the advantages/disadvantages and main differences between both extensions.

Summing up, the primary contribution of this article is to report the experience we have gained from creating both a LW and a HW Aspect-Oriented Extension of KDM. The secondary contribution is the experiment we have conducted whose goal was to answer the following Research Question (RQ):

> RQ – Which of the KDM AOP extensions (LW or HW) requires less effort (time) and leads to less errors when creating and maintaining their instances?

In the following, we present the background related to ADM and KDM, extension alternatives for KDM and aspect-oriented modernization scenario. Then, in Section 3 we present the aspect-oriented extensions of KDM. After, in Section 4 we discuss the evaluation of the approach. In Section 5 we discuss about threats to validity of our research. In Section 6 we describe some related works, Section 7 presents the lesson learned herein, and finally in Section 8 we draw some conclusions and describe plans for future work.

2. ADM & KDM

Architecture-Driven Modernization (ADM) is a trend of reengineering processes that considers standard metamodels and MDA concepts (like PIM, PSM and CIM) along the process. According to OMG, the main reason of this problem is the lack of standardization, hindering the productivity of teams, preventing the reuse of algorithms and techniques and also compromising the interoperability among modernization tools from different vendors (Ulrich and Newcomb, 2010; Sadovykh et al., 2009).

The modernization process supported by ADM involves three phases and it is similar to a horseshoe (Kazman et al., 1998): (*i*) reverse engineering, (*ii*) restructuring, and (*iii*) forward engineering, as can be seen in Fig. 1. Starting from the lower left side, in reverse engineering part, the knowledge is extracted from legacy systems and PSM is generated. The PSM is used as a base to generate a PIM that conforms to an ADM metamodel named Knowledge Discovery Metamodel. After obtaining the PIM, one can generate the CIM going up to the level of abstraction. Thus, during reverse engineering, transformations are done aiming to get a high-level representation of the system, independently of the adopted platform.

In restructuring phase, it is possible to conduct refactoring (Durelli et al., 2017, 2014a), optimization (Landi et al., 2017; Chagas et al., 2016), and also insert new business rules in the system. Please note that this restructuring phase can be performed in any level of the horseshoe (PSM, PIM and CIM level). The output is a new target model without the problems previously identified, which can be called “a modernized model” in any level of the horseshoe.

In the sequence, we can proceed to the forward engineering phase, wherein the models are resubmitted to a set of transformations to reach the source-code level again.

The PIM and CIM abstractions can be represented by the main ADM metamodel, called Knowledge Discovery Metamodel (KDM)[2]. The KDM is a metamodel of common intermediate representation to existent systems and its operating environments. Using this representation it is possible exchange systems representation between platforms and languages aiming to analyze, to standardize, and to transform existing systems (OMG, 2016).

The KDM can represent physical and logical software artifacts in different abstractions levels and it is formed by twelve packages organized in four layers: (*i*) infrastructure, (*ii*) program elements, (*iii*) runtime resources, and (*iv*) abstractions. In Fig. 2 it is shown the KDM architecture with its layers (right side) and the internal packages, which can also be seen as sub-metamodels because each package represents a different system’s view (Normantas et al., 2012).

[2] Formal specification of KDM: https://www.omg.org/spec/KDM/About-KDM/.

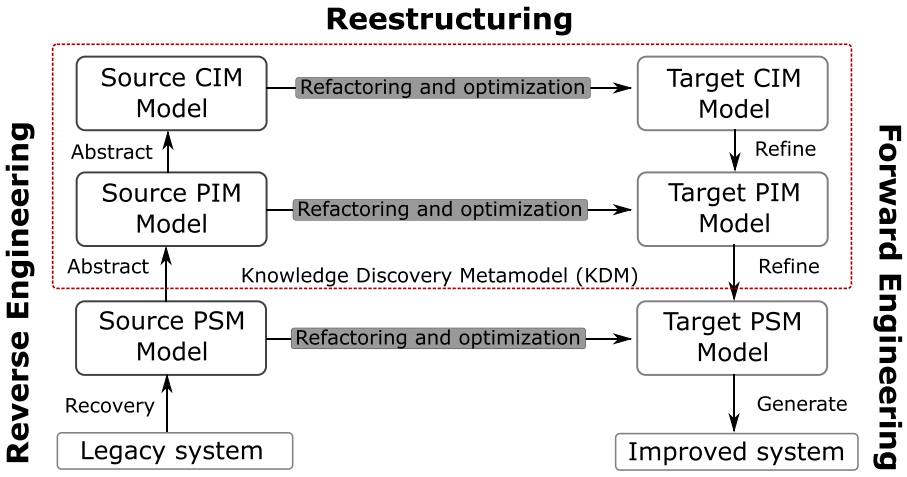

Fig. 1. Process flow of modernization supported by ADM (Pérez-Castillo et al., 2011).

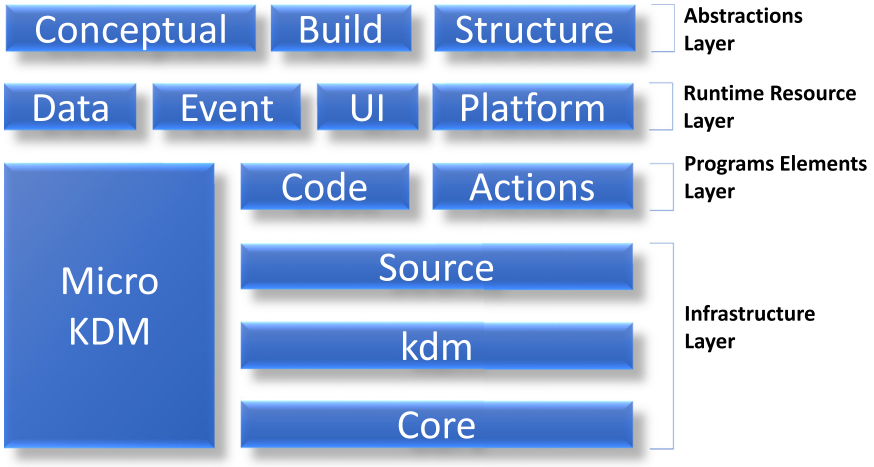

Fig. 2. KDM's architecture. Adapted from Pérez-Castillo et al. (2011).

In this paper, the main goal is to represent the AOP concepts in KDM. To develop a Heavyweight (HW) KDM extension that represents the AOP concepts it is necessary extend some metaclasses from Code package, located in programs elements layer. Regarding the Lightweight (LW) extension, the package to be used is the Kdm Package from infrastructure layer.

2.1. Code package

The Code package defines a set of metaclasses, whose purpose is to represent implementation-level program units and their associations. The package also includes metaclasses that represent common program elements supported by various programming languages, such as: data types, classes, procedures, macros, prototypes, and templates.

In a given instance of KDM, each element of the Code package represents some construct in a programming language, determined by the programming language used in the system. The Code Package consists of 24 classes and contains all the abstract elements for modeling the static structure of the source code.

In Table 1 is depicted some of them. This table identifies KDM metaclasses possessing similar characteristics to the static structure of the source code. Some metaclasses can be direct mapped, such as class and interface from OO language, which can be easily mapped to the `ClassUnit` and `InterfaceUnit` metaclasses from KDM.

Table 1
Mapping between code elements and the KDM Metaclasses.

Code element	Metaclass
Class	`ClassUnit`
Interface	`InterfaceUnit`
Method	`MethodUnit`
Attribute	`MemberUnit/StorableUnit`
Parameter	`ParameterUnit`
Association	`KdmRelationship`

2.2. Kdm package

Kdm package describes several infrastructure elements that are present in each KDM instance. Together with the elements defined in the Core package these elements constitute the so-called KDM Framework. The remaining KDM packages provide meta-model elements that represent various elements of existing systems.

Kdm package is a collection of classes and associations that define the overall structure of KDM instances. From the infrastructure perspective, KDM instances are organized into segments and

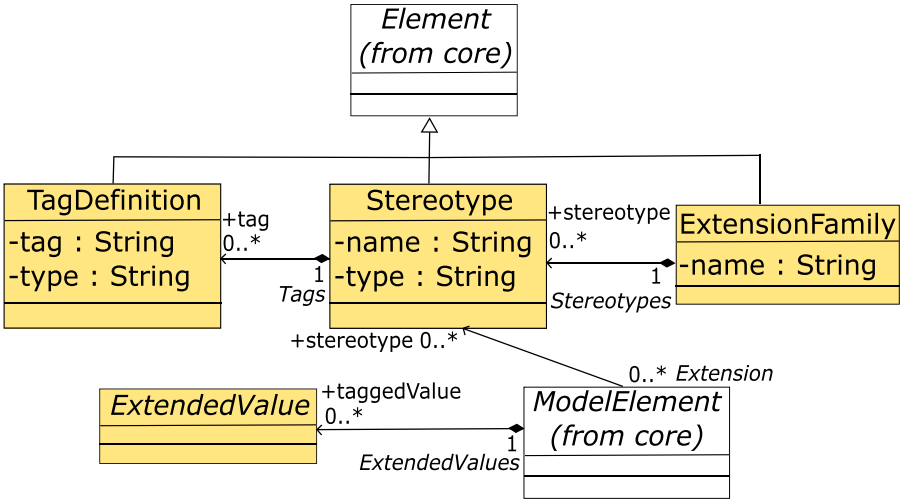

Fig. 3. Lightweight extension metaclasses.

then further into specific models. Kdm package consists of the following five class diagrams: (*i*) Framework – defines the basic elements of the KDM framework, (*ii*) Audit – defines audit information for KDM model elements, (*iii*) Annotations - provides user-defined attributes and annotations to the modeling elements, (*iv*) Extensions - a class diagram that defines the overall organization of the lightweight extension mechanism of KDM, and (*v*) ExtendedValues - the tagged values used by the lightweight extension mechanism. We have used the last two class diagrams to create the lightweight extension mechanism presented herein.

2.3. Extension alternatives for KDM

As already stated there are two ways of extending KDM: (*i*) lightweight (LW) and (*ii*) heavyweight (HW). In the following subsections we detail each of them.

2.3.1. Lightweight extensions

The KDM has a package called "Kdm" that involves a set of metaclasses for creating lightweight extensions by means of stereotypes and tagged values. Part of the class diagram of Kdm Package can be seen in Fig. 3.

The `ExtensionFamily` metaclass acts as a container for encapsulating a set of related stereotypes. The `Stereotype` metaclass represents stereotypes, which are ways of annotating metaclass instances so that they can represent a concept different from the original meaning. The `TagDefinition` metaclass represents the stereotype tags, which are used for adding attributes in the stereotypes. The `ExtendedValue` metaclass defines common properties to `TaggedValue` and `TaggedRef` and represents the value of an attribute.

The precise meaning of each new stereotype is defined out of the KDM scope and it should be informed by the developers so the users could properly use the extended representations.

The LW KDM extension mechanism neither allow tags multiplicity, tags constraints, nor relationship between tags and stereotypes. Thus, the engineer responsible for the extension creation should choose the most specific metaclass to define the stereotype with the semantics between the element to make sure that the stereotype make sense.

2.3.2. Heavyweight extensions

Heavyweight extensions consists of creating (or modifying the existing ones) new metaclasses and incorporate them in the metamodel. Most of the time, the new metaclasses extend the existing ones. Usually, heavyweight extensions are much more expressive than lightweight ones, but they hinder the reusability of the metamodel instances.

The creation of a heavyweight KDM extension does not require the existence of a specific package, as occurs in the lightweight version. It is just necessary to create new metaclasses of modifying existing ones.

Besides, one can devise metaclasses in any KDM package, i.e., one can devises new metaclasses in the Code package, or in the Structure package, etc.

In the context of this work, the Code KDM package is the central package for the heavyweight extension we have created. This happens because all the concepts we have created were implementation concepts thus the only package that these new concepts could fit was Code package.

2.4. Aspect-Oriented modernization scenario

According to Pérez-Castillo and Piattini (2011) there are several modernization scenarios that can be conducted to modernize legacy systems: Platform Migration, Application Improvement, Non-Invasive Application Integration, Data Architecture Migration, Service-Oriented Architecture Transformation, Language to Language Conversion, and Paradigm to Paradigm migration.

The scenario we are dealing herein is the last one since we are coping with modernizations from Object-Oriented (OO) system to Aspect-Oriented (AO) ones. It is important to highlight that, in this scenario, it is not mandatory changing the language, i.e., it is possible to convert OO systems to AO versions using the same language. Although not common and, possibly this is not the best alternative, it can be performed using dependency injections and other alternative strategies. The most normal way is to use an aspect-oriented language, like AspectJ (Kiczales et al., 1997).

The aspect-oriented modernization scenario we are working with is shown in Fig. 4. In the left lower part, we can find a legacy system with modularization problems, i.e., there are some cross-cutting concerns (light gray, dark gray, and black) bad modularized as they are spread throughout the system modules. The modernization goal is to get an aspect-oriented version in which the modularization problems are solved. This is represented by the target system, in which the concerns are now well modularized, as can be seen in the low right part.

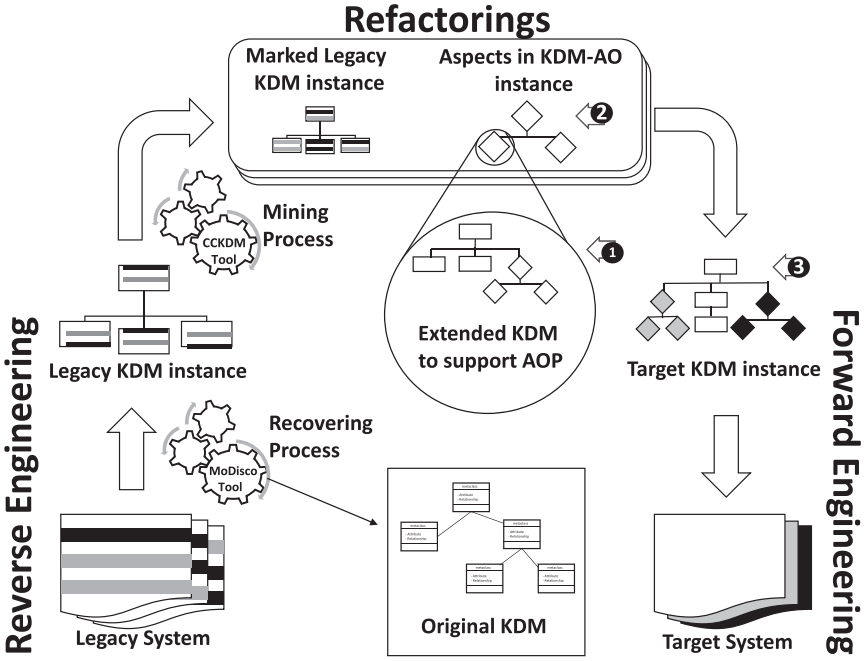

Fig. 4. Aspect-oriented modernization scenario.

As any other modernization scenario, it starts by reverse engineering the system into a KDM instance that represents the system as is, i.e., a representation of the system with the same spreading and scattering problems presented in the source code. This model will be called here as "Legacy KDM".[3]

As soon as the legacy KDM is recovered, a concern mining process it is needed for identifying the source code elements (present in the KDM model) that contribute to the implementation of the concerns. This mining process also needs to annotate these elements in the KDM. This process is represented as a gear and the output is the annotated KDM. An example of a tool that can be used in this step is the CCKDM[4], proposed by Martín Santibáñez et al. (2015).

In the sequence, the restructuring phase gets the annotated KDM as input and performs aspect-oriented refactorings on this model. The output is a new modernized KDM instance with the concerns modularized with aspect-oriented abstractions. The goal of the restructuring phase is to analyze the annotated elements of the legacy KDM and creating aspect-oriented abstractions that allow a better modularization of those concerns.

In the scenario presented in Fig. 4, an AO KDM Extension is needed for representing the output of the restructuring phase. This occurs because the AO refactorings get OO elements and need to write AO ones, thus, the AO abstractions must be available in this target model. This situation happens with any other modernization process that reads specific element and needs to write/create a different element, which it is not present in the original version of the metamodel. Therefore, as can be seen in the figure, the original version of KDM supports the phases prior the restructuring and the AO Extended version supports the activities after the restructuring.

The icon represented by an arrow and a circle shows when the AO concerns are used. The first time shows the modified KDM with aspect-oriented concepts, the second time represents some refactorings using the KDM-AO metamodel, and the third time illustrates the target KDM instance that is the refactored system in KDM model. Note that the others parts of the figure are not treated in this paper. Also note that the "Recovering Process" is executed by an existing tool called *MoDisco*[5] (Bruneliere et al., 2010) and the "Mining Process" is executed by CCKDM tool that was previously developed by Martín Santibáñez et al. (2015).

3. Aspect-oriented extensions of KDM

This section presents the two KDM extensions we have developed - the lightweight and the heavyweight. The creation of these two KDM extensions had as the starting point an UML profile for AOP proposed by Evermann (2007). Evermann's profile is a well accept and used profile in the academic area but we also considered other approaches to compose ours (Soares et al., 2002; Rausch et al., 2003; Júnior et al., 2010). One of the distinguishing characteristics of the AO UML profile proposed by Evermann is the completeness. It covers most of the AspectJ elements, concentrating not only on the basic concepts, like Aspects, Pointcuts and Joinpoints, but also the different types of pointcuts (PreInitialization Pointcuts, WithintCode Pointcut, etc) and intertype declarations.

Please note that both extension presented herein are devised in meta-level. More specifically, in the heavyweight extension new metaclasses (representing aspect-oriented concepts) are added in the metamodel. These new metaclasses (representing the AO concepts) extend existing KDM metaclasses. In lightweight extension, there is no inclusion of new metaclasses or modifications of existing ones in the metamodel. The extension here is done by means of the creation of stereotypes and tagged values, but it is also in the meta-level, since the set of stereotypes created are available for all KDM instances.

In Fig. 5 there is a class diagram that presents both extensions we have created – the light and the heavy one. This picture has been adapted from Evermann (2007) and it is used here to represent both the new metaclasses we have created in the heavyweight extension and also the new stereotypes of the lightweight extension.

[3] Please note that in this section, we use double quotes (" ") to reference the elements in the Fig. 4.

[4] Available in: https://github.com/dsanmartins/cckdm.

[5] Available in: https://www.eclipse.org/MoDisco/.

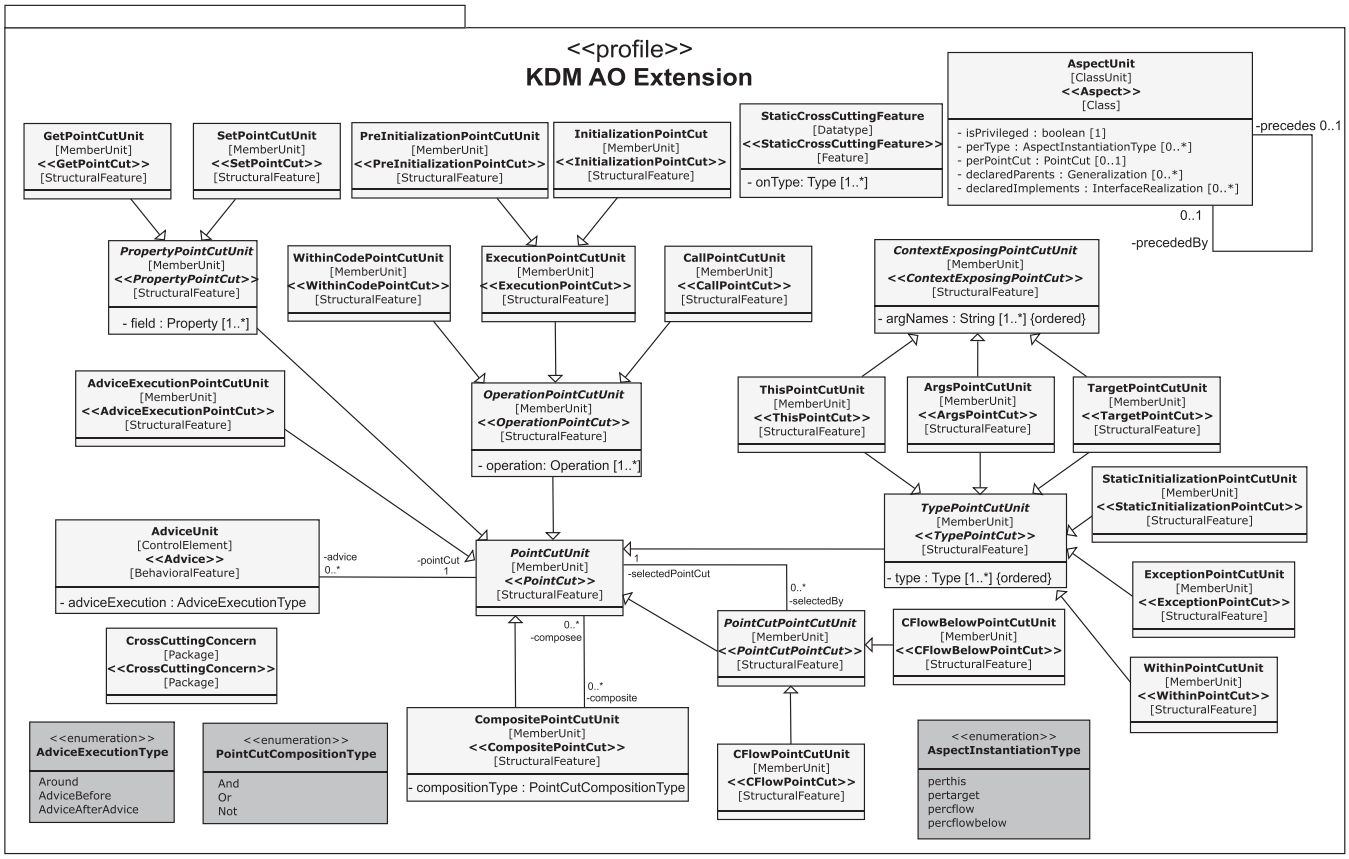

Fig. 5. Light and heavyweight extensions. Adapted from Evermann (2007).

As can be seen in Fig. 5, each class has four lines in the first compartment. The first word represents the name of the metaclass we have created in heavyweight extension. For example, `AspectUnit` is a new metaclass we have created in the heavyweight extension. The second word, in brackets, is the KDM metaclass that was chosen to serve as the base class of the current element. For example, the new `AspectUnit` metaclass extends the `ClassUnit` metaclass from KDM (Santos et al., 2014b).

In lightweight extensions there is not addition of new metaclasses, just the creation of stereotypes. In order to represent this in Fig. 5, there is a `<<stereotype>>` symbol in the third line of the element. Therefore, all stereotypes shown in Fig. 5 exist in the lightweight extension. Stereotypes only can be applied on existing metaclasses, thus the name of the element that the stereotype can be applied to is presented in second line. For example, the stereotype `<<aspect>>` can only be applied in `ClassUnit` instances.

In order to provide a different visualization of the extensions, the Table 2 shows the complete list of metaclasses (heavy) and stereotypes (light) created. The first column represents all the metaclasses created in KDM metamodel to materialize the heavyweight extension and the second column represents all the stereotypes of the lightweight extension. The third column represent the original KDM metaclasses used as base for both extensions.

One of the biggest challenges when extending metamodels is to choose which metaclass is the most suitable one. As the stereotypes of Evermann's profile had already been previously mapped to UML metaclasses, we decided to take a deeper look and analyze if this information could be useful. As there are some similarities between UML and the KDM Code Package, the information was useful. However, to systematize this analysis, we built a mapping between both metamodels (UML and KDM), that can be seen in Table 3.

This mapping shows a semantic correspondence between KDM and UML metaclasses. In some cases, the mapping is straightforward, such as `Class` from UML and `ClassUnit` from KDM. They have the same goal of representing classes in an object-oriented context. However, as KDM can represent higher and lower abstraction levels than UML, some UML metaclasses do not have just one candidate in KDM. Besides, there are others that have no equivalent. The `Property` UML metaclass, for example, has three possible candidates metaclasses in KDM: `StorableUnit`, `ItemUnit`, and `MemberUnit`. `StorableUnit` represents primitive type variables, `ItemUnit` represents records, and `MemberUnit` represents associations with others classes.

This semantic gap happens because the KDM code package is in a lower abstraction level than UML. There are also KDM metaclasses that do not have corresponding metaclasses in UML, in consequence of the low abstraction level. For example, the `CodeAssembly` metaclass is a metaclass that represents a logical element container, written in machine language, that were build in a specific operating system or hardware. There is no UML metaclass for representing this concept.

Table 2 we can see the existing relation between the metaclasses and also comments about them. As KDM is a metamodel broader than UML, many relations consider only the code package from KDM, because this package is the only one that can represent classes, attributes, methods, relationship and others elements with static features. Other KDM packages are concentrated in other dimensions that are also present in UML 2.0, such as user interfaces, architecture, and conceptual abstractions, but KDM was designed to support the modernization process. To attend to one of

Table 2
Aspect-oriented LW and HW mapping elements.

AO metaclasses (Heavyweight)	AO stereotypes and tags (Lightweight)	Base KDM metaclass
AspectUnit	aspectUnit	ClassUnit
PointCutUnit	pointCutUnit	MemberUnit
CompositePointCutUnit	compositePointCutUnit	MemberUnit
OperationPointCutUnit	operationPointCutUnit	MemberUnit
WithinCodePointCutUnit	withinCodePointCutUnit	MemberUnit
ExecutionPointCutUnit	executionPointCutUnit	MemberUnit
CallPointCutUnit	callPointCutUnit	MemberUnit
PreInitializationPointCutUnit	preInitializationPointCutUnit	MemberUnit
InitializationPointCutUnit	initializationPointCutUnit	MemberUnit
PropertyPointCutUnit	propertyPointCutUnit	MemberUnit
GetPointCutUnit	getPointCutUnit	MemberUnit
SetPointCutUnit	setPointCutUnit	MemberUnit
AdviceExecutionPointCutUnit	adviceExecutionPointCutUnit	MemberUnit
PointCutPointCutUnit	pointCutPointCutUnit	MemberUnit
CFlowPointCutUnit	cFlowPointCutUnit	MemberUnit
CFlowBelowPointCutUnit	cFlowBelowPointCutUnit	MemberUnit
TypePointCutUnit	typePointCutUnit	MemberUnit
WithinPointCutUnit	withinPointCutUnit	MemberUnit
ExceptionPointCutUnit	exceptionPointCutUnit	MemberUnit
StaticInitializationPointCutUnit	staticInitializationPointCutUnit	MemberUnit
TargetPointCutUnit	targetPointCutUnit	MemberUnit
ArgsPointCutUnit	argsPointCutUnit	MemberUnit
ThisPointCutUnit	thisPointCutUnit	MemberUnit
ContextExposingPointCuitUnit	contextExposingPointCuitUnit	MemberUnit
CrossCuttingConcern	crossCuttingConcern	Package
SetAdviceExecution	adviceExecutionType (tag)	TaggedValue
SetPointCutCompositionType	pointCutCompositionType (tag)	TaggedValue
SetAspectInstantiationType	aspectInstantiationType (tag)	TaggedValue
StaticCrossCuttingFeature	staticCrossCuttingFeature	Datatype
AdviceUnit	adviceUnit	ControlElement

Table 3
Mapping UML - KDM.

UML	KDM	Comments
Class	ClassUnit	The metaclass Class from UML intends to represent the same concept of the ClassUnit metaclass from KDM. The metaclass Class (UML/ Basics package) has four properties: isAbstract, ownedProperty[*], ownedOperation[*] and superClass. The ClassUnit element, from Code Package encompasses all of these properties through the AbstractCodeElement class. A ClassUnit may have any attribute whose type is a concrete class of AbstractCodeElement, like StorableUnit, MemberUnit, ItemUnit, MethodUnit, CommentUnit, KDMRelationships, etc.
Operation	MethodUnit	The semantic of the Operation metaclassfrom UML is closer to the MethodUnitmetaclass from KDM. This happens beauseOperation (UML/Basics package) is a behavioralelement that has the following properties: class(specifies the owner class), ownedParameter(Operation's parameters) and raisedException(Operation's exceptions). The MethodUnit class is the ideal element to represent Operations because itis a behavioral KDM element capable to representthe most diverse programming languages operations.MethodUnit has attributes like kind (defines the kindof the operations, for example: abstract, constructor,destructor, virtual, etc.) and export (defines the accessmodifiers, for example: public, private and protected).
Property	StorableUnit; ItemUnit; MemberUnit	Property (UML) represents variables in general (local variables, global variables, arrays, associations, etc.),while KDM has an element for each kind of Property: primitive type variable (StorableUnit), records and arrays (ItemUnit) class members (MemberUnit).
Package	Package	A Package in UML (Basics package) is very similarly to a KDM Package (Code Package). Both are containers for program elements, like classes, and others code elements. A Package could have one or more classes, and a class could have many others elements, like methods, properties, comments, etc.
StructuralFeature	DataElement	StructuralFeature (UML/Core::Abstractions package) is an abstract metaclass that can be specialized to represent a structural member of a class, like a property. The KDM has the DataElement class (Code package), that can be specialized to StorableUnit, MemberUnit or ItemUnit.
BehavioralFeature	ControlElement	BehavioralFeature (UML/Core::Abstractions package) is an abstract metaclass that can be specialized to represent behavioral members of a class. The equivalent class on KDM is the ControlElement, an abstract class that can be specialized to represent callable elements, including behavioral elements like MethodUnit.
Parameter	ParameterUnit	Parameter (UML/ Core:Abstractions) is an abstract metaclass to represent the name and the type of the element that will be passed by parameter in a behavioral element. On the KDM we can use the ParameterUnit class. This metaclass can also represent the name, type, position of the parameter in the signature and the kind of parameter (value or referece).
Relationship	KDMRelationship	Both Relationship and KDMRelationship metaclasses are abstract metaclasses that can be specialized to represent some kind of relationship between two elements, like Aggregation, Generalization, etc.

```
1 ExtensionFamily AspectConcepts = KdmFactory.eINSTANCE.createExtensionFamily
      ();
2 Stereotype AspectUnit = KdmFactory.eINSTANCE.createStereotype();
3 AspectConcepts.getStereotype().add(AspectUnit);
4 AspectUnit.setName("AspectUnit");
5 AspectUnit.setType("ClasUnit");
6 TagDefinition IsPrivileged = KdmFactory.eINSTANCE.createTagDefinition();
7 AspectUnit.getTag().add(IsPrivileged);
8 IsPrivileged.setTag("isPrivileged");
9 IsPrivileged.setType("boolean");
10 [...]
```

Listing. 1. Creating the AspectUnit stereotype.

our goals, this mapping table shows only the main elements that were used in the aspect-oriented KDM extensions (KDM AO Extension), once the full mapping of the ninety metaclasses from code package would be infeasible. Nevertheless, all the classes from Evermann's profile were mapped and are represented in Table 3. In our website[6] we provide others mapping tables developed by us.

3.1. The lightweight AO extension

This subsection shows how we have created the Lightweight AO extension (Santos et al., 2014a). In this research the LW extension was created programmatically by using Java language in the Eclipse IDE (Integrated Development Environment). As previously commented, KDM provides a set of metaclasses in a package called kdm that allows the creation of stereotype families, stereotypes, and tagged values, as shown in Fig. 3. Stereotype families are a kind of container for a light weight extension.

In Listing 1 is shown part of the whole source code of the LW KDM AO Extension. In this listing, only the source code for creating the AspectUnit stereotype is shown. It is possible see the creation of three instances in Kdm Package: `ExtensionFamily`, `Stereotype` and `TagDefinition`. In line 1 it is shown the creation of an instance of `ExtensionFamily` element named `AspectConcepts`. This element encapsulate all the created stereotypes to the lightweight KDM AO profile. In the second line an instance of `Stereotype` element is shown and it is possible to see the creation of `AspectUnit` stereotype. Once a stereotype is created it is necessary specify the `ExtensionFamily` that it belongs to. The source code snippet presented in line 3 adds the stereotype created in line 2 inside the `ExtensionFamily` element created in line 1.

Lines 4 and 5 are filled with `Name` and `Type` values of the stereotype, which are `String` type. In this line the `setName` value is `AspectUnit` and represents the name of the stereotype, differently of what occurs in line 2 that the name `AspectUnit` represents an instance of the `Stereotype` element.

Line 6 the `TagDefinition IsPrivileged` is created and in line 7 this tag is attached to the `AspectUnit` stereotype. In line 8 and 9 the `Tag` and `Type` properties of `TagDefinition` element are defined. Once more, the filled values of these elements are `Strings`, as is defined by the KDM rules.

All the stereotypes, relationship and attributes shown in Fig. 5 were programmatically added and properly attached, i.e., the stereotypes were attached to an `ExtensionFamily` and the relationships and the attributes were attached to their respective stereotypes. Once all the elements were programmatically created it was possible to reuse them by means of a Java class with all the programmed `Stereotypes` and `TagDefinitions`.

3.2. The heavyweight AO extension

The procedure for building the heavyweight extension was to create a new KDM metaclass for each stereotype of Evermann's profile (Santos et al., 2014a). The main difference is the base metaclass used; instead of using the same UML metaclass used by Evermann, we used our mapping table (see Table 3) to find an equivalent in KDM. For example, if a stereotype in Evermann's profile extended the Class metaclass of UML, in our heavyweight extension the new metaclass should extended the `ClassUnit` of KDM, as these classes are equivalent in these metamodels.

As can be seen in Fig. 5, the main aspect-oriented elements from Evermann's profile are represented as higher level classes/stereotypes: `CrossCuttingConcern`, `Aspect`, `Advice`, `Pointcut` and `StaticCrossCuttingFeature`. The remaining elements are subclasses.

For example, `CrosscuttingConcernUnit` is a new metaclass we have created for representing the existence of a crosscutting concern, such as persistence, security and concurrence. In Evermann's profile this element extends the Package metaclass from UML. In KDM AO extension this element extends the Package metaclass from KDM. This KDM metaclass represents a standard package where it is possible to group aspects, classes and others elements from AO and OO programming languages.

`AspectUnit` is a new metaclass for representing an aspect and it extends the `ClassUnit` metaclass. The decision to extend the `ClassUnit` metaclass is justified because this element has all the characteristics that an aspect can have, besides, it can support new elements such as `Pointcuts`, `Advices` and inter-type declarations.

`AdviceUnit` is a new metaclass for representing advices. The element to represent advices is `AdviceUnit`, that extends the `ControlElement` metaclass. Knowing that advice is an element that specifies behavior, it is possible to consider an advice as a method. Nevertheless, advices do not have neither access specifiers (public, private and protected) nor types (constructor, destructor, etc.). Because of this were decided not to make `AdviceUnit` metaclass extend the `MethodUnit` behavior.

[6] http://advanse.dc.ufscar.br/index.php/research-projects/fapesp-2017.

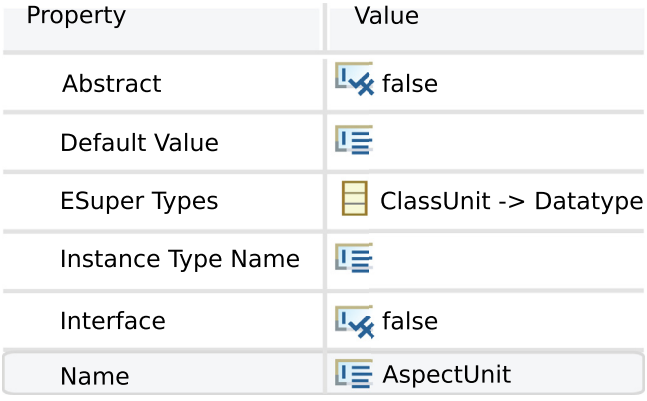

Fig. 6. `AspectUnit` element properties.

`PointCutUnit` is a new metaclass for representing `Pointcuts` and join points. According to Evermann's profile, `PointCut` is a structural element and extends a `StructuralFeature` metaclass from UML. The KDM also has a metaclass to represent structural features called `DataElement`, which is an abstract metaclass. Their sub-metaclasses are `StorableUnit`, `MemberUnit`, and `ItemUnit`. As a `PointCut` can be abstract, and the `StorableUnit` and `ItemUnit` metaclasses can not, `MemberUnit` were chose to be the super-metaclass of `PointCutUnit`. Besides, other motive that influenced in the choice of using the `MemberUnit` as super-metaclass was the fact that the `Pointcuts` crosscuts others classes, and the `MemberUnit` is the KDM metaclass that is used to make references to members of others classes inside of a determinate class.

`StaticCrossCuttingFeature` is new metaclass for representing inter-type declaration. In heavyweight extension this element can extend two metaclasses: `StorableUnit` and `MethodUnit`. Thus, `StaticCrossCuttingFeature` is capable of representing not only structural features but also behavioral features. Therefore, a `StaticCrossCuttingFeature` instance can be an attribute or a method that will be inserted in a determinate class.

`CrossCuttingConcern` is a new element that was extended from `Package` element of KDM Code Package. However, none attribute or additional relationship were inserted in the new element, once its creation aimed only separate the concerns in a KDM model, without adding behavior.

Creating KDM Extensions. To create the heavyweight KDM AO extension version the Eclipse IDE and the Framework Eclipse Modeling Framework (EMF) were used. This allowed the edition and visualization of the original KDM in ".ECORE" format. More details of this process can be seen in Santos et al. (2014b).

Summarizing, by means of these tools we could insert all the metaclasses depicted in Fig. 5. In Fig. 6 the `AspectUnit` metaclass properties are shown. As we can see, the element created has some properties that have to be filled, such as `Abstract`, `ESuperTypes` and others. Each new metaclass has its properties set in different ways, but it has to be in accordance to the proposed profile (KDM AO extension). After the creation of all new metaclasses in KDM, the heavyweight KDM AO extension plug-in version were created, allowing the creation of new aspect-oriented instances of KDM.

In Listing 2 an instantiation example of the heavyweight KDM AO extension is presented. In line 1 an instance of AspectUnit metaclass is created, in line 2 it is informed the name of the instance and in line 3 the property IsPrivileged is filled.

Regarding the reuse of heavyweight KDM AO extension plug-in one should say that it can only be reused in Eclipse tool, once it was developed in this IDE. However, the AOP mataclasses, properties, and relationship inserted in the original KDM, can be reused in any tool that could read XMI source-code from ".ECORE" extension. By reading the XMI information of the new metaclasses and by creating a tool/plug-in with these new information, we believe that it would be possible to create AO KDM instances independently of the programming language (AspectC++, AspectS, etc.), because the metaclasses here created are extensions of the existing classes, that are platform and language independents.

4. Evaluation

In this section we present an experiment to investigate whether software engineers productivity (time and errors) is different when using HW and LW KDM extensions. In this experiment, the subjects had to create and change (apply maintenance modifications) instances of the both KDM AO extensions we have developed. All the planning was done according to Wohlin guidelines (Wohlin et al., 2000).

As already pointed out in Section 1 our RQ is:

> RQ – Which of the KDM AOP extensions (LW or HW) requires less effort (time) and leads to less errors when creating and maintaining their instances?

4.1. Method

4.1.1. Participants

The experiment was conducted in the context of a Software Engineering course of the Federal University of São Carlos (UFSCar) in Brazil. The participants (subjects) were 14 graduate candidates in computer science.

4.1.2. The aspect-oriented framework

In order to simulate a real situation, we had to choose some aspects, pointcuts and joinpoints to be modeled by the subjects. Therefore, instead of creating hypothetical elements we decided to use an existing AO persistence framework (de Camargo and Masiero, 2008), because it had almost all existing aspect of aspect-oriented elements. Thus, all the activities performed by the subjects involved the creation and the maintenance of aspects, pointcuts and joinpoints of this framework.

4.1.3. Selection of variables

The dependent variables are the instantiation time, the maintenance time and the number of errors. Instantiation time is the time subjects spent to create instances of the extended KDM. Maintenance time is the time subjects spent for modifying (change) existing KDM-AO instances. The number of errors represents the number of errors a subject got in each activity.

The independent variables or factors are:

1. *KDM AO extensions:* There are 2 levels for this factor; the HW and LW extensions.
2. *Activities performed by the subjects:* There are 6 levels for the developing activities and 2 levels for the maintaining activities. In Table 4, we present the description of the activities.

4.1.4. The planning of the experiment

The experiment was planned in blocks (Wohlin et al., 2000), to ensure the subject's experience did not interfere in the result. Table 5 presents the experiment organization.

```
1 AspectUnit myAspect = CodeFactory.eINSTANCE.createAspectUnit();
2 myAspect.setName("connectionComposition");
3 myAspect.setIsPrivileged(true);
```

Listing. 2. HW extension instance example.

Table 4
Activities performed by the subjects.

Development activities	
Activity number	Activities description
1	Creating three different CrosscuttingConcerns (security, logging and persistence)
2	Creating three different Aspect and associate them with the Crosscutting Concerns created in activity 1.
3	Creating three different PointCut with a joinpoint each.
4	Creating two different PointCut with two joinpoints each.
5	Creating three different Advice and link them to the PointCuts created in activity 3.
6	Creating five different Inter-Type Declaration.
Maintenance activities	
Activity number	Activities description
7	Adding three properties in a specific Aspect.
8	Transforming a PointCut with a Joinpoint in a PointCut with two Joinpoints.

Table 5
Groups distribution in relation to the extensions.

	Group 1 (7 subjects)	Group 2 (7 subjects)
Phase 1	LW AO KDM	HW AO KDM
Phase 2	HW AO KDM	LW AO-KDM

In Phase 1, the groups 1 and 2 worked in parallel - while group 1 performed the activities (development and maintenance) using the LW extension, group 2 used the HW one. Once they have finished the activities, they where allowed to proceed to the second phase, where the extensions were shift between the groups.

Each group was submitted to both factors, the extensions and the activities,but in different phases. This was done to verify if the order impacts the result, i.e., if instantiating an specific KDM extension prior to the another one lead to different results.

As shown in Table 5, all the activities performed by the subjects were related to the same context - a persistence framework. The activities involved the creation of `aspects`, `Pointcuts`, `join points`, `Advices` and `Intertype declarations` of this persistence framework.

The performed activities were divided into development and maintenance activities. The name of each instance were informed to the subjects during the experiment. The development activities are concentrated on creating new KDM AO elements, while the maintenance ones are focused on changing existing AO KDM instances. The complete activities can be seen in https://github.com/Advanse-Lab/KDM-AO.

4.2. Operational steps of the experiment

The experiment was performed in three steps: (*i*) preparation, (*ii*) execution, and (*iii*) data validation. We explain these steps in the following subsections.

4.2.1. Preparation of the experiment

In this step, the materials to be used in the experiment were elaborated[7].

[7] The artifacts used in the experiment are available in the link: https://github.com/Advanse-Lab/KDM-AO.

4.2.1.1. Instrumentation. The following documents were developed to be used in the experiment: (*i*) Subjects Characterization Form, to get the professional experience and in the topics related to the study; (*ii*) Consent Form, to subjects approval and consent of the study objectives and the participation terms; (*iii*) Description of the Activities with the instructions of its execution; (*iv*) Guide for Using the LW and HW Extensions; (*v*) Mapping table of the AspectJ elements to LW and HW extensions and (*vi*) Class diagram of the extensions, so the subject could know which attributes and relationships belong to a determinate element.

4.2.1.2. Data collecting instruments. A data collecting form was elaborated to gather data, in which the subjects should fill all the required information during the experiment execution. In the same form, there was a field for qualitative evaluation, so each subject should report its perception about the difficulties, easiness and suggestions while using the extension. This was done after they have finished the experiment. This form was elaborated in the same file that the activities descriptions of the experiment, so the subject could access all the information and could record their conclusion times in the same document.

4.2.1.3. Training and pilot. All the subjects were submitted to training and pilot sessions prior the real experiment. In the training we explained about AOP and how to create KDM instances using the LW and HW extensions. The training took four hours in total; two hours for explaining about the main topics and two hours for exercising. The goal was to make them proficient in the creation of KDM-AO instances (LW and HW). In this day we also handed the consent and characterization forms to the subjects, so that this information could be used in the pilot.

In the pilot's day, we simulated the activities that would be performed in the real experiment. These activities helped us on improving some details, such as the time limit required for each activity and the way the activities should be distributed. The pilot was organized in groups, the 14 subjects were divided in two equal groups and each group should use both approaches, but in different phases.

All the subjects used the same application, i.e., they should create aspects with generic names and without context, for example, `Aspect A`, `Pointcut pt1`, etc. This was different from the

Table 6
Subjects distribution according to their punctuation.

Punctuation (Points based on their skills)					
Subject	Very low 0 a 10	Low 11 a 20	Normal 21 a 30	High 31 a 40	Very High 41 a 50
1			24		
2			21		
3			27		
4					41
5		19			
6			21		
7	9				
8			24		
9			29		
10				32	
11					42
12		20			
13	10				
14				35	
Total	2	2	6	2	2

real experiment where the created aspects were related to the AO persistence framework. The training and pilot day were conducted in such way that in the end all the subjects could use both approaches.

In the real experiment day we basically used the same steps of the pilot; the main difference were: (*i*) the activities and their descriptions were delivered to the subjects in a formal document, instead of explaining them by means of a presentation, and (*ii*) all of the activities that should be done were explained in the beginning of the experiment, so this could improve the subjects' time and avoid time interruptions.

Once the subjects have started the activities, they could only solve their doubts with the delivered artifacts. Any other doubt they had should be registered in the qualitative evaluation form.

Another difference between the experiment and the pilot was the activities categorization. We categorize the activities in "development" and "maintenance". We decided to make this differentiation to investigate not only the productivity in writing new KDM instances but also the productivity in maintaining the existing ones.

4.2.2. Execution of the experiment

Firstly, the subjects were positioned in the groups based on its punctuation of the Subjects Characterization Form. Each group had seven subjects. In Table 6 the subjects punctuation are represented considering the form questions and the subjects total to each punctuation category. Both groups had the same subjects quantity in the same category, i.e., each group had one subject with very low punctuation (0 to 10), one subject with low punctuation (11 to 20), three subjects with average punctuation (21 to 30), one subject with high punctuation (31 to 40) and one subject with very high punctuation (41 to 50). After the subjects were allocated, they received all the artifacts needed to perform the activities.

Regarding the data of the experiment, for some activities our interest was the time spent for conducting it. In this case, the time was registered in minutes. In other cases, our interest was in the number of errors and, in this case, it was analyzed typos, missing punctuation and omitted reserved words. Thus each correction performed in a code line in order to make it works properly was counted as one error. For instance, if a statement contains one typo and two missing reserved words the errors amount is 3 (three). For information purposes, all tables were obtained by using the **R** statistical software.

Table 7
Means and standard deviation for time variable of development activities.

Technique	Activity	Time.mean (min)	Time.sd (min)
HW	Actv-1	2.571429	0.7559289
HW	Actv-2	3.642857	1.0082081
HW	Actv-3	5.000000	1.1766968
HW	Actv-4	6.714286	1.4898927
HW	Actv-5	4.857143	2.5975474
HW	Actv-6	3.642857	1.2157393
LW	Actv-1	3.428571	1.0894096
LW	Actv-2	4.285714	0.6112498
LW	Actv-3	7.571429	0.8516306
LW	Actv-4	16.571429	2.9277002
LW	Actv-5	9.357143	2.2051389
LW	Actv-6	5.214286	1.2513729

4.3. Data validation of development activities

4.3.1. On time variable

In this section we analyze the data of the 14 subjects who performed the six development activities and the effect of these two factors (extensions and activities) on the time variable. The type of analysis that was conducted is called as two-within subjects factors (two-way repeated measures ANOVA), because each group performs all the development activities of the two extensions.

In Table 7, we present the means (Time.mean) and standard deviations (Time.sd) of the time variable related with development activities and in Fig. 7 the corresponding box-plot. We can see that box-plots of the HW extension are more or less homogeneous in comparison with the box-plots of LW extension. Box-plots of LW.Actv-4 and LW.Actv-5 look like they have the largest amount of spread data for LW extension, indeed Table 7 corroborate that impression because the standard deviation values of each one are 2.92 and 2.20 respectively. Thus, this is an indicator that these activities requires more attention to be analyzed.

In order to use multifactorial ANOVA for analyzing the data, a precondition imposed is the assumption of normality. Therefore, our first analysis was checking whether there is no violation of normality by using Shapiro–Wilk test. The test showed that the data are not normal. The complete analysis can be found in the following URL (https://github.com/Advanse-Lab/KDM-AO). In order to overcome this problem we apply a non-parametric test for multiple factor with repeated measures: the Aligned Rank Transform Procedure (ART) (Wobbrock et al., 2011).

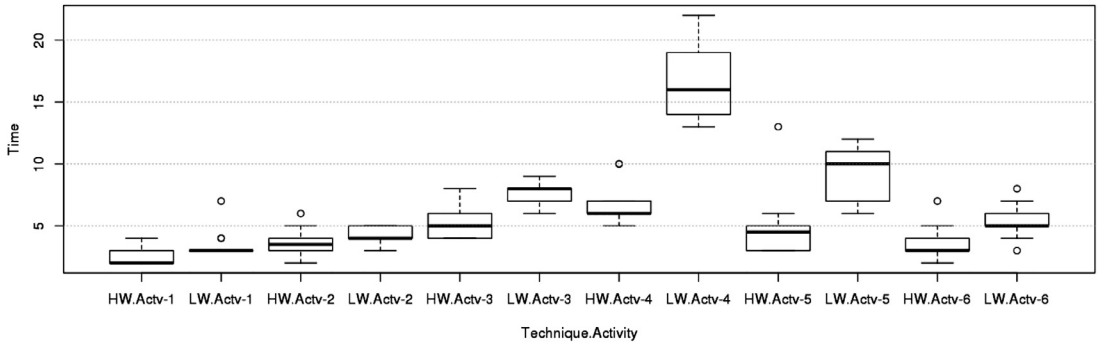

Fig. 7. Average time of development activities on time variable.

Table 8
ART test of development activities on time variable.

		F	*Df*	*Df.res*	*Pr*(> *F*)	Signif. codes
1	Technique	243.092	1	143	< 2.22e-16	***
2	Activity	129.659	5	143	< 2.22e-16	***
3	Technique:Activity	48.669	5	143	< 2.22e-16	***

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1 Model: Mixed Effects (lmer), Response: art(Time). F: Statistic. Df: Degree of Freedom. Df.res: Residual degrees of freedom. Pr(> F): Significance probability.

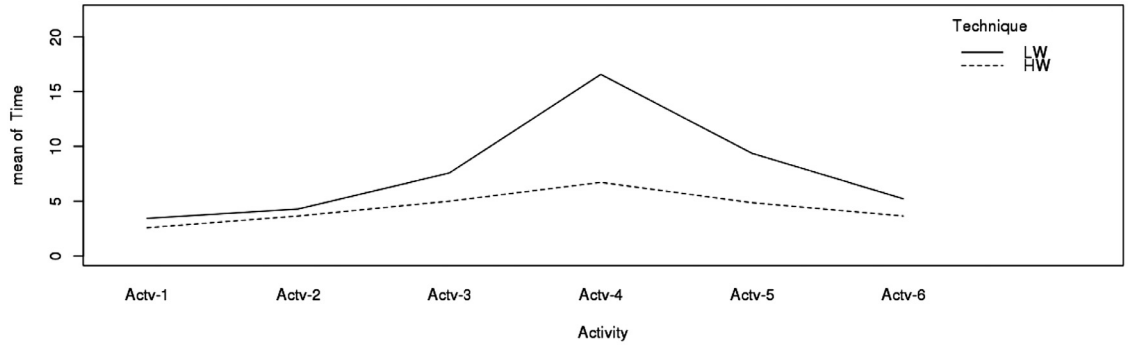

Fig. 8. Interaction plot of development activities on time.

In Table 8, we show the results of the application of the ART test on the development activities, where the *p*-values for each factor and the interaction between them are statistically significant, meaning there is a main effect of extensions on time, a main effect of activities on time and a significant interaction between extensions and activities. A "main effect" is the effect of a single independent variable on a dependent variable, in this case, the effect of extension and activity on time variable.

We can also see graphically the interaction between the two factors by means of an interaction plot. In Fig. 8 we show the interaction plot of development activities between the two extensions (LW and HW) on time. The line that represents the LW extension always holds a gap in relation with the line that represents the HW extension. That means there is a main effect of extension on time because LW always take more time than HW for developing the activities.

From Actv-1 to Actv-2 the lines are sloped and almost parallel so there is a main effect on activity because both require slightly more time. From Actv-2 to Actv-3 there is a behaviour called "alligator jaws", in Actv-2, both of the extensions are basically the same, very close in performance but in Actv-3, the LW extension technique has now become deferentially worst than HW extension technique in terms of developing time. The same can be said from Actv-3 to Actv-4 and Actv-4 to Actv-5. Finally, from Actv-5 to Actv-6 both techniques decrease the development time but the gap between the lines is bigger than the gap between Actv-1 and Actv-2.

In Table 9 we show the pairwise comparison among the activities of development by taking into account the time variable. This can be interpreted by posing the following questions: Is the difference between HW and LW significantly different in condition Actv-1 to condition Actv-2?. No, because its *p*-value $>.05$. Is the difference between HW and LW significantly different in condition Actv-2 to condition Actv-3?. Yes, because its *p*-value $<.05$. Is the difference between HW and LW significantly different in condition Actv-3 to condition Actv-4?. Yes, because its *p*-value $<.05$. Is the difference between HW and LW significantly different in condition Actv-4 to condition Actv-5?. Yes, because its *p*-value $<.05$. Is the difference between HW and LW significantly different in condition Actv-5 to condition Actv-6?. Yes, because its *p*-value $<.05$.

4.3.2. On error variable

In this section we analyze the data of the fourteen subjects which performed six development activities by using the two extension techniques in KDM models, LW and HW and the effect of these two variables (technique and activity) on the error variable.

Table 9
Interaction contrast of development activities on time variable.

		Value	*Df*	*Chisq*	*Pr(>F)*	Signif. codes
1	HW-LW : Actv -1-Actv -2	−3.857	1	0.0573	0.810797	
2	HW-LW : Actv -2-Actv -3	57.857	1	12.8951	0.001977	**
3	HW-LW : Actv -3-Actv -4	146.214	1	82.3553	< 2.2e-16	***
4	HW-LW : Actv -4-Actv -5	−82.429	1	26.1738	2.496e-06	***
5	HW-LW : Actv -5-Actv -6	−90.643	1	1.6504	1.661e-07	***

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1. Chisq Test (χ^2), *p*-value adjustment method: holm. Df: Degree of Freedom. Df.res: Residual degrees of freedom. Pr(> F): Significance probability.

Table 10
Means and standard deviation for error variable of development activities.

Technique	Activity	Error.mean	Error.sd
HW	Actv-1	0.0000000	0.0000000
HW	Actv-2	0.2142857	0.8017837
HW	Actv-3	0.1428571	0.5345225
HW	Actv-4	0.2142857	0.8017837
HW	Actv-5	0.0000000	0.0000000
HW	Actv-6	0.5000000	1.6052798
LW	Actv-1	0.0000000	0.0000000
LW	Actv-2	0.0000000	0.0000000
LW	Actv-3	0.0000000	0.0000000
LW	Actv-4	0.8571429	1.8337495
LW	Actv-5	0.3571429	0.9287827
LW	Actv-6	0.4285714	1.6035675

Table 11
ART test of development activities on error variable.

		F	*Df*	*Df.res*	*Pr(>F)*	Signif. codes
1	Technique	10.7276	1	143	0.0013243	**
2	Activity	1.8368	5	143	0.1093124	
3	Technique:Activity	3.2235	5	143	0.0086730	**

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1. Model: Mixed Effects (lmer), Response: art(Error). F: Statistic. Df: Degree of Freedom. Df.res: Residual degrees of freedom. Pr(> F): Significance probability.

The type of analysis that was conducted in this section is the same as the one we made in the previous section.

In Table 10, we present the statistics of means and standard deviations of error variable related with development activities for HW and LW techniques. We do not provide the corresponding box-plot because there are several zero values in the standard deviation column that does not bring valuable information to our analysis. Instead of that, an interaction plot could be more useful to analyze main effects and interactions among levels of factors.

As in the previous analysis, herein we also check whether there is not a violation of normality by using Shapiro–Wilk test. The test showed that the data are not normal and the complete analysis can be found in the following URL (https://github.com/Advanse-Lab/KDM-AO). Thus, we apply the non-parametric test for multiple factor with repeated measures ART.

In Table 11, we show the results of the ART test of development activities on error, where the *p*-values are just significant for technique and the interaction between technique and activity. That means that there is an overall main effect of the technique on error and a significant interaction between technique and activity.

In Fig. 9 we show the interaction plot of development activities between the two techniques on error. As we see, the lines of the two techniques are crossing and that is a classic picture of interaction effect. Also, if we draw a line between the two techniques the proportion areas seems to be similar which indicates, in overall, there is not much changes in activity, as the ART test indicated previously (*p*-value > .05).

Table 12 shows the interaction contrast of development activities on error variable. Line 6 presents the only significant *p*-value, that means there is significant differences on error by using HW and LW extension techniques when developers perform Actv-3 or perform Actv-4.

Some conclusions can be made from the statistical analysis. The first one is that developers tend to make more errors when they are using the LW technique than HW technique. The second one is that the activity does not affect significatively the quantity of errors. The third one is that techniques in combination with Act-3 and Act-4 have an effect on errors. One explanation could be that the time for developing these activities is higher than the others because of the difficulties involved, and so more errors may be introduced in the development.

4.4. Data validation of maintenance activities

4.4.1. On time variable

In this section we analyze the data of the fourteen subjects which performed two development activities by using the two extension techniques in KDM models, LW and HW and the effect of these two variables (technique and activity) on the time variable. The type of analysis that was conducted is called as two within subjects factors (2 repeated measures factors), because each group of the seven subjects perform all the maintenance activities of the two techniques.

In Table 13, we present the statistics of means and standard deviations of time variable related with maintenance activities for HW and LW techniques and in Fig. 10 the corresponding box-plot of the data.

It seems that there is no big differences in time when applying the HW technique to both activities. Similarly, there is no big differences in time when applying the HW technique to both activities. Nevertheless, HW technique performs better (in less time) than LW technique.

As in the previous analysis, herein we also check whether there is not a violation of normality by using Shapiro–Wilk test. The test showed that the data are normally distributed. Thus, we analyze it by using a parametric test called Linear Mixed Model (lmm) which belongs to the lme4 package (Bates et al., 2015) of the **R** statistical software. The complete analysis can be found in the following URL (https://github.com/Advanse-Lab/KDM-AO).

Table 14 shows lmm test of maintenance activities on time variable. There are main effects on Technique and on Activity but there is not an interaction of these factors. Indeed, Fig. 11 shows the interaction plot of maintenance activities where the lines of each technique are slightly parallel without crossing between them, consequently with the result of lmm test analysis.

In light of the overall significant result, we can do some pairwise comparisons among the levels of technique and activity. In Table 15, we show the result of these comparisons and it presents

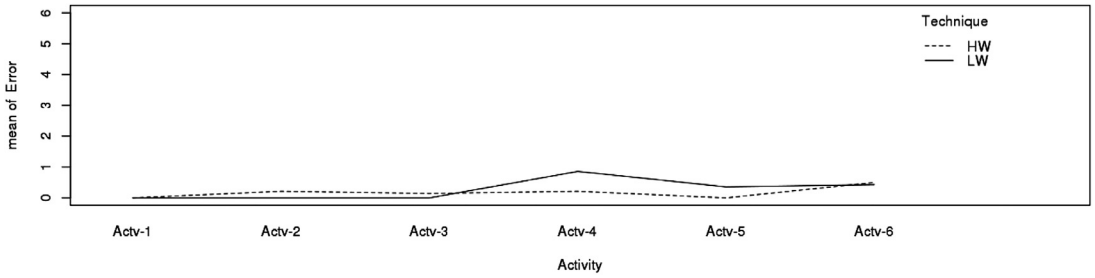

Fig. 9. Interaction plot of development activities on error.

Table 12
Interaction contrast of development activities on error variable.

		Value	*Df*	*Chisq*	*Pr(> F)*	Signif. codes
1	HW-LW : Actv-1-Actv-2	−18.357	1	1.7922	1.00000	
2	HW-LW : Actv-2-Actv-3	3.143	1	0.0525	1.00000	
3	HW-LW : Actv-3-Actv-4	41.571	1	9.1913	0.03404	*
4	HW-LW : Actv-4-Actv-5	−11.214	1	0.6689	1.00000	
5	HW-LW : Actv-5-Actv-6	−20.071	1	2.1426	1.00000	

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1. Chisq Test (χ^2), *p*-value adjustment method: holm. Df: Degree of Freedom. Df.res: Residual degrees of freedom. Pr(> F): Significance probability.

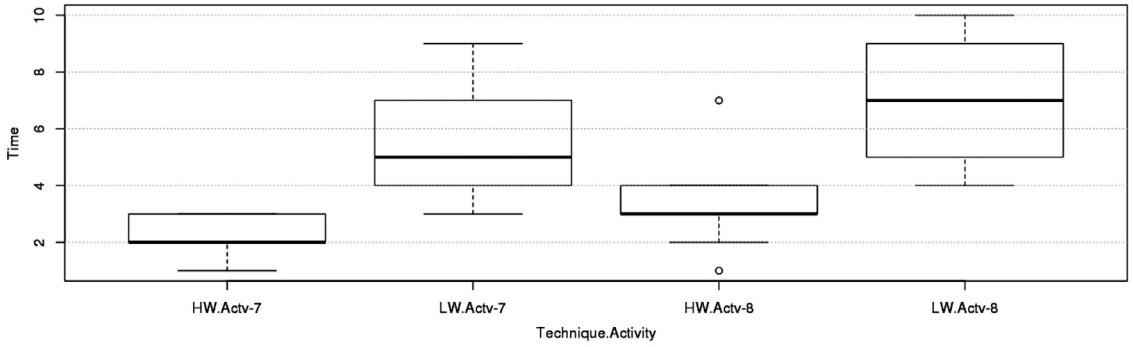

Fig. 10. Average time of maintenance activities on time variable.

Table 13
Means and standard deviation for time variable of maintenance activities.

Technique	Activity	Time.mean	Time.sd
HW	Actv-7	2.214286	0.6992932
HW	Actv-8	3.428571	1.3424596
LW	Actv-7	5.500000	1.6984156
LW	Actv-8	7.142857	2.0701967

all the pairwise comparisons available across of the levels in the interaction plot. Note that all *p*-values are significant, so there are differences on time when using different techniques for different maintenance activities.

4.4.2. On error variable

In this section we analyze the data of the fourteen subjects which performed two development activities by using the two extension techniques in KDM models, LW and HW and the effect of these two variables (technique and activity) on the error variable.

In Table 16, we present the statistics of means and standard deviations of error variable related with development activities for HW and LW techniques. We do not provide the corresponding boxplot because of the values of the data it does not bring valuable information to our analysis. Instead of that, an interaction plot could

Table 14
Main LMM test of maintenance activities on time variable.

		F	*Df*	*Df.res*	*Pr(> F)*	Signif. codes
1	(Intercept)	246.5185	1	13	7.849e-10	***
2	Technique	109.1364	1	39	7.313e-13	***
3	Activity	18.1818	1	39	0.0001234	
4	Technique:Activity	0.4091	1	39	0.5261708	

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1. Model: Mixed Effects (lmer), Response: art(Error). F: Statistic. Df: Degree of Freedom. Df.res: Residual degrees of freedom. Pr(> F): Significance probability.

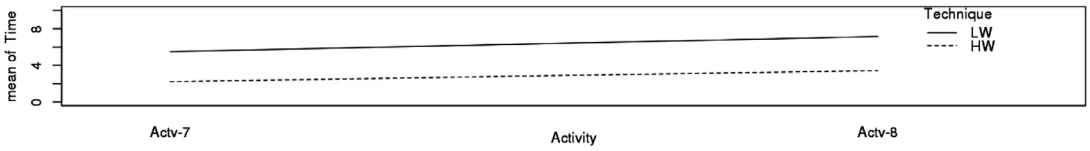

Fig. 11. Interaction plot of maintenance activities on time.

Table 15
Post hoc pairwise comparisons of maintenance activities on time variable.

Simultaneous tests for general linear hypotheses					
	Est.Std.	*Error*	*t value*	*Pr(>|t|)*	Signif. codes
HW,Actv-7 - LW,Actv-7 == 0	−3.2857	0.4738	− 6.935	1.05e-07	***
HW,Actv-7 - HW,Actv-8 == 0	−1.2143	0.4738	− 2.563	0.014355	*
HW,Actv-7 - LW,Actv-8 == 0	−4.9286	0.4738	−10.402	4.97e−12	***
LW,Actv-7 - HW,Actv-8 == 0	2.0714	0.4738	4.372	0.000266	***
LW,Actv-7 - LW,Actv-8 == 0	−1.6429	0.4738	− 3.467	0.002591	**
HW,Actv-8 - LW,Actv-8 == 0	−3.7143	0.4738	− 7.839	7.77e−09	***

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1. *p*-value adjustment method: holm. Fit: lme4::lmer(formula = Time ∼ (Technique * Activity) + (1 | Subject)). Pr(> |t|): Significance probability.

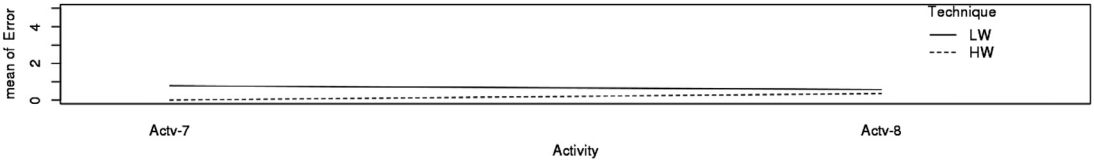

Fig. 12. Interaction plot of maintenance activities on error.

be more useful to analyze main effects and interactions among levels of factors.

As in the previous analysis, herein we also check whether there is not a violation of normality by using Shapiro–Wilk test. The test showed that the data are not normal and the complete analysis can be found in the following URL (https://github.com/AdvanseLab/KDM-AO). Thus, we apply the non-parametric test for multiple factor with repeated measures ART.

In Table 17, we show the results of the ART test of maintenance activities on error, where there are not overall significant main effects on technique, activity and also there is not an interaction between the two factors because p-values are $>.05$ in all cases. In Fig. 12 we show the interaction plot of maintenance activities between the two techniques on error. Lines are slightly separated in activity Actv-7 and tend to join in activity Actv-8. Indeed, for activity Actv-7 there were no errors when developers used HW technique and means of error for activity Actv-8 when developers used LW technique is very low. Thus, the gap between the two lines is negligible and therefore, we state that there is not a main effect on technique.

In Table 18, we show the interaction contrast of maintenance activities on error variable. There is not significant differences on error by using HW and LW extension techniques when developers perform Actv-7 or perform Actv-8.

4.5. Discussion of results

After analyzed statistically the results, we can make some conclusions about the extension techniques for the KDM. The analysis performed in Section 4.3 shows that in the overall, the LW technique performed worse than HW technique, for all development activities. However, it was notorious the poor performance of developers in activities 3, 4 and 5 with the LW technique. It seems that activities that requires the extension of several AOP structures with the LW technique, such as the combination of PointCuts and JoinPoints, take more time to implement because developers need to write more lines of code in comparison with the HW extension technique.

The analysis in Section 4.3.1 shows that in the overall, the usage of LW or HW extension technique for developing activities imply in the rise of errors. This is in compliance with the previous analysis because as developers write more lines of code is reasonable that they make more mistakes in the codification.

The analysis in Section 4.4 shows that in the overall, the LW technique performed worse than HW technique, for the two maintenance activities. However, both of the techniques increase the maintenance time in the activity Actv-8. The activity Actv-8 deals with PointCut and JoinPoints that is the same as in the previous scenario, taking more time to be implemented by developers.

Finally, the analysis in Section 4.4.2 shows that in general the techniques and the activities does not affect significantly the number of errors because few errors were committed by developers.

5. Threats to validity

As with any experimental study, this experiment has several threats to validity. In this section, we consider the study of (Cook and Campbell, 1979) as a template to discuss the threats that might jeopardize the validity of our experiment. Internal validity is concerned with the confidence that can be placed in the cause-effect relationship between the treatments and the dependent variables in the experiment. External validity has to do with generalization, namely, whether or not the cause-effect relationship between the treatments and the dependent variables can be generalized outside the scope of the experiment. Conclusion validity focuses on the conclusions that can be drawn from the relationship between treatment and outcome. Finally, construct validity is about the adequacy of the treatments in reflecting the cause and

Table 16
Means and standard deviation for error variable of maintenance activities.

Technique	Activity	Error.mean	Error.sd
HW	Actv-7	0.0000000	0.0000000
HW	Actv-8	0.3571429	0.8418974
LW	Actv-7	0.7857143	1.0509023
LW	Actv-8	0.5714286	1.3985864

Table 17
ART test of maintenance activities on error variable.

		F	*Df*	*Df.res*	*Pr(> F)*	Signif. codes
1	Technique	2.1128	1	39	0.15407	
2	Activity	1.5888	1	39	0.21499	
3	Technique:Activity	1.5733	1	39	0.21720	

Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1. Model: Mixed Effects (lmer), Response: art(Error). F: Statistic. Df: Degree of Freedom. Df.res: Residual degrees of freedom. Pr(> F): Significance probability.

the suitability of the outcomes in representing the effect. We categorized all threats to validity according to this classification.

5.1. Internal validity

We mitigated the experience level of participants by splitting all the participants in two balanced groups. To the creation of these two groups we have considered the experience level based on Table 6 and we have balanced the groups considering the total points of each subject.

The training phase was focused on presenting the AOP concepts, the metamodel extensions mechanisms and how to create KDM-AO instances (LW and HW). Thus, no training on Model-Driven Architecture (MDA) or ADM were done. However, the subjects (masters and PhDs candidates) already had a preparation by the professor of the course, so we did not have to be concerned about it. Another point is that as they were instantiating the metamodel extensions programmatically, the model-based part was abstracted.

Another internal validity is the productivity under evaluation. There is a possibility that this might influence the experiment results because students often tend to think they are being evaluated by experiment results. In order to mitigate this, we explained to the students that no one was being evaluated and their participation was considered anonymous. However, we cannot rule out the possibility that some participants has been influenced by this threat.

5.2. External validity

The sample might not be representative of the target population. As mentioned, we carried out the experiment with 14 subjects, which were divided into two group. We cannot rule out the threat that the results could have been different if another sample had been selected. However, to diminish this threat we have performed the training stage in order to provide to the subjects the knowledge in extension mechanisms needed the make the sample the most representative.

It is possible that the exercises are not accurate for every maintenance's problem for real world applications. To mitigate this threat, the activities were designed considering applications based on the real world.

Another point is that the activities were application-independent as can be seen in Table 4. For example, creating three pointcuts is not different if the pointcuts is from application *Y* or *X*, because just the name of them is different. The main point is that creating elements in the LW extension demands more lines of code and, consequently, more effort and error proneness.

Table 18
Interaction contrast of maintenance activities on error variable.

		Value	*Df*	*Chisq*	*Pr(> Chisq)*
1	HW-LW : Actv -7-Actv-8	-8	1	1.5733	0.2097

Chisq Test (χ^2), *p*-value adjustment method: holm. Df: Degree of Freedom. Pr(> Chisq): Significance probability.

5.3. Conclusion validity

The main threat to conclusion validity has to do with the quality of the data collected during the course of the experiment. We have evaluated the performance of the subjects considering the time to perform the activities and the number of errors in each task. About the time, we have asked to the subjects to set the start and the end time. In this sense we could have had a problem because a subject could forget to mark the time in the form. To mitigate this, we have set a standard start time to all subjects and to certificate that they were recording the time in each activity we had three monitors in the room just to check this.

About the numbers of errors, each subject had to hand two set of files, one to each treatment, the challenge was to catch all the errors without miss a single one. To mitigate this, we have performed the error checking by two experts from our group, if the number of errors of a subject was the same we considered right, if not we would have to check again until the number of errors be the same.

5.4. Construct validity

The subjects already knew the researchers and they also knew that the HW instantiation process was supposed to be easier (less source code to be written) if compared to the LW one. Both of these issues could affect the collected data and cause the experiment to be less impartial. In order to avoid impartiality, we enforced that the participants had to keep a steady pace during the whole study and that both approaches had their advantages and disadvantages.

Since we have created both KDM-AO extensions we claim that we had no preference neither for the LW one nor for the HW one. Thus we have eliciting the main advantages and disadvantages of both in this paper.

6. Related works

This section presents the related works of our approch. We split this section in three topics: (*i*) specific approaches to KDM extensions, (*ii*) generic approaches such as UML's extensions and (*ii*) approaches that use aspect-oriented in legacy systems.

6.1. Specific approaches - KDM extensions

The work more related to ours is the KDM AO extension created by Shahshahani (2011). As we have done here, this author also created a heavyweight KDM extension for aspect-oriented programming. There are three main differences between our works. Firstly, while Mirshams has based her extension on an aspect model created by herself, we have created our extension based on a very well known profile for aspect-oriented programming. Evermann's profile encompasses all the AO concepts presented in AspectJ and in other aspect-oriented languages, like Aspect C++ and AspectS.

The second difference is the scope of our extensions. The aspect model used by Mirshams contains much less elements than

Evermann's profile. That means our extension is able to represent both a high level (using the most generic metaclasses) and a low level (using most specific metaclasses) view of the system. In her case, just a higher level view is possible. The third difference is that her work is limited to dynamic crosscutting as there are no elements for representing inter-type declarations. However, despite all of these differences, the main similarity is that we have used the same KDM metaclasses she has used too.

Another KDM extension is presented by Baresi and Miraz (2011). They proposed a heavyweight KDM extension to support Component-Oriented MOdernization (COMO). COMO is a metamodel that supports traditional concepts of software architecture, allowing to attach software components in KDM. Using their extension it is possible to replace or add parts of a system. Unlike we have done here, in their paper they had not used an existing profile as the starting point for creating their extension - they combined another metamodel to the KDM. COMO extends some high level metaclasses of KDM, such as `KDMModel`, `KDMEntity` and `KDMRelationship`. These classes are the base of their extension and provide the link between KDM and COMO metamodels.

The main similarity with our work is that they have also performed a heavyweight extension in KDM. As a main difference, the extension presented by them only extended high level elements of KDM, while in our solution we have use more specific elements such as `ClassUnit` and `MemberUnit`.

Usually event logs are represented with particular notations such as Mining XML (MXML) rather than the recent software modernization standard, such as KDM. Therefore, (Pérez-Castillo et al., 2012) created an extension aiming to mitigate this limitation, i.e., the authors have extended KDM's Event model to describe event logs. Similarly to our lightweight KDM extension they also have used the `ExtensionFamily` mechanism – allowing them to create `Stereotypes` comprising different `TagDefinitions` and also permitting them to integrate event logs in KDM, tagging the extracted entities with the extended concepts.

6.2. Generic approaches - UML extensions

Similar to our work and the work proposed by Shahshahani (2011) there are researches that seek to perform AOP extensions in UML (Ahmed et al., 2017; Zakaria et al., 2002; Qaisar et al., 2013; Stein et al., 2002).

Ahmed et al. (2017) focuses on creating a lightweight UML extension which supports language specification for AspectJ. Similar to our extension they also have used Eclipse IDE – more specifically they used Eclipse Modeling Framework (EMF), which is the core to represent KDM's metamodels in Eclipse.

Zakaria et al. (2002) proposed an UML extension for modeling AO system. The authors used the lightweight mechanism to create the AO UML extension. Lightweight UML extension mechanism are based on "Stereotypes", "Tagged Values", and "Constraints". As presented in Listing 1 our lightweight AO extension mechanism also contains "Stereotypes" and "Tagged Values" (known in KDM's metamodel as "TagDefinition"), see lines 2 and 6, respectively. Zakaria et al., proposed different types of tags for relationship between the classes and aspects, differently in our approach one just need to call the method `createTagDefinition()` and add this tag into a "Stereotypes", see Line 6 and Line 7 in Listing 1.

Qaisar et al. (2013) describe a metamodel for AOP in which they proposed an extension for AOP. The authors used Meta Object Facility (MOF) which has the heavyweight extensibility mechanism in its specification. Likewise the KDM, the authors tried to create a complete metamodel, i.e., according to the authors, the proposed metamodel not only models the static structure of AOP but also can models the behavioral structure of the model. The authors also defined new metaclasses, for example, they define the following metaclasses: `Aspect`, `PointCut`, and `Advice` – these metaclasses are strongly similar to our heavyweight extension mechanism: `AspectUni`, `PointCutUnit`, and `AdviceUnit`.

Stein et al. (2002) also create AO extension to design notation for AspectJ programs. Similar to our approach the authors classified similarities between UML elements and AspectJ's features. The extension proposed is used in three UML's diagrams, class diagram, use case diagram, and sequence diagram.

6.3. Aspect-oriented approaches in legacy systems

The approaches of Schutter and Adams (2007) and Chen et al. (2010) are focused on reengineering legacy systems with the help of aspect-orientation. The approache of Schutter and Adams (2007) developed a method to generate class and sequence diagrams using techniques of reflection and decompilation from the Java binary byte code of AO legacy systems. The authors based their approach on the Java Reflection and decompiler tools.

The approach proposed by Chen et al. (2010) address the combination of AO programming and meta-programming during the revitalization of legacy systems in Cobol and C. To address these combinations, the authors used four use cases that are: (1) reverse engineering; (2) recovery of business logic; (3) encapsulation of business applications for integration with service-oriented environments; and (4) maintenance and bug-fix of legacy systems. The authors achieved relevant results for the first three use cases where AO programming and meta-programming showed that can aided for the address problems, but for the last use case the AO programming solution present too much of a limitation for Cobol legacy systems although the problem can be managed reasonably for C legacy systems.

Although these publications are model-driven approaches and their goals are in showing the reengineering/modernization process of legacy systems our work is more focused on showing how the modernization process could benefit from KDM aspect-oriented extension.

7. Lessons learned and limitations

This section discusses the lessons learned of our investigation and shows some limitations of modernization processes based on ADM.

The first lesson learned is that there is a lack of ready-to-use modernization tools that could help in the validation process of a modernization scenario. In conducting this research, we learned that the power of ADM is strongly influenced by the ability to represent specific concepts in an appropriate way. For instance, a major concern in reverse and forward engineering steps derives from the heterogeneity of how to represent software systems, in which the data are often uniformly represented as many models. Therefore, ADM by means of its standards aims at switching from the heterogeneous scenario to the homogeneous scenario. The big idea is to retrieve one or several models from a given system, depending on the needed viewpoints, and then to work directly on these models.

Nowadays there are very few tools that give full support to reverse and forward engineering based on ADM scenario. As far as we know, there are few initiatives to provide more generic integrated reverse engineering tools that can be extended and used to different scenarios. A tool that we have used herein is *MoDisco*. Although *MoDisco* is used in this project, its core components support discovery just for Java technologies, i.e., there is a lack of tools that provides fully support for other implementation technologies such as C/C++, C# (.NET) or COBOL. In fact, Clause (Clausen, 2012)

devised a Python discovery based on ADM. However, we could not find online the source-code to test this discovery.

Thus, after developing our AO-extensions and performing the validations that we have shown in this paper, the ideal scenario would be to perform a real modernization project using the extensions but this was not possible because we neither had an aspect-oriented discoverer nor a forward engineering tool to convert KDM models in source-code again.

The second lesson learned is the importance of a research that presents how to extend KDM in a light and heavyweight manner. As we present in the related works section, there is a shortage of guidelines on how to extend KDM as well as lack of criteria on how to compare the instantiation process of HW and LW extensions.

ADM claims KDM can represent all software artifacts, however, sometimes it is needed to represent specific domain concepts and that is why there are the extension mechanisms. With the conduction of our research, we could notice that the choice for an extension (HW or LW) will depend on the purpose that it will be used for and for each one there is a set of consequences to be considered. For instance, the LW extension mechanism is less demanding, since its creation process requires less effort and its reuse is more easily adapted in existing tools. In general terms, the main advantage of using the HW extension is due to the quality assurance of the produced instances. Another advantage of using the LW extension mechanism is the speed and convenience of adding new behaviors in instances of KDM, since its creation process is less labor intensive if compared to the HW mechanism.

As a third lesson learned we claim that there is a lack of research about the synergy between KDM and others ADM standard metamodels. It is noted that KDM is a powerful metamodel that can be adapted to several domains, so it will only depend on the software engineer to create a solution that best serves its purpose. However, KDM is a metamodel to represent software systems artifacts and it does not provide a graphical visualization of its content. For this purpose, a modernization engineer should use another metamodel such as UML or Business Process Model and Notation (BPMN), depending on the required point of view.

In other research of our group (Durelli et al., 2017), we evaluated the application of refactorings in KDM instances with the support of UML classes diagrams in an experiment involving seven systems implemented in Java, by using a tool named Knowledge Discovery Model-Refactoring Environment (KDM-RE). The systems used in the experiment were Xerces-J, Jexel, JFreeChart, JUnit, GanttProject, ArtofIllusion, and JHotDraw. According to the authors, these seven systems were chosen because they are real-world Java applications whose sizes range from 16,026 to 240,540 lines of code. In spite of the fact that, KDM seems to be a robust metamodel to represent complete systems it is not possible to state that the results can be generalized for all Java applications instantiated using KDM and represented in UML classes diagrams.

As a forth lesson learned we claim that dealing with the number of errors has brought us a better understanding and knowledge on how to create automated support for the creation of KDM-AO instances. We claim that this knowledge have came from two sources: the definition of what would be considered as an error and the importance of counting the number of errors.

Regarding the first part of the sentence, we have defined what would be an “error” so that we could count the number of errors the subjects committed. We believe it is not so important the granularity of the error, but how we are counting them.

Regarding the second part, that is the importance of counting the number of errors, we believe there are two important points: i) Clearly, the most expected situation would be to have an automated tool for creating the instances of the KDM extensions. However, the Modernization Engineer, in charge of implementing such an automated support, must create “scripts” that automate the creation of instances. The subjects of the experiment played the role of these scripts and this was very useful for identifying the most frequent errors in the process of creating AO KDM instances. The second point is that, doing such an exercise of creating the instances manually, we learned all the steps for the correct implementation of the scripts.

8. Conclusion

In this paper we presented our investigation on aspect-oriented extensions for the Knowledge-Discovery Metamodel (Santos et al., 2014b). To conduct this investigation we have developed a heavyweight and a lightweight extension and conducted an experiment that evaluated the productivity when creating instances of these extensions and also when modifying these instances. The main goal is to deliver the KDM AO extensions created so that Aspect-Oriented Modernization Projects can be conducted.

The experiment has concentrated on analyzing the productivity when creating instances (and also modifying them) of these both KDM AO extensions. The tasks were performed programmatically for the software engineers and an AO Persistence Framework was employed (de Camargo and Masiero, 2008). The statistical analysis showed that the HW and LW do have impact over the time and also the number of errors when creating the instances. So the choice between these options must be carefully analyzed. Summarizing, when HW extension is employed, the software engineers are 43% faster than using LW extensions. Besides, the software engineers commit 7.7% less errors if compared to the subjects that have used the LW extension.

Regarding the effort for creating the extensions, we claim that it is quite similar. The creation of a LW extension requires the instantiation of some classes of the kdm package for creating stereotypes and tagged values, but none new metaclass is created. Similarly, in the HW case, one must create new metaclasses that must be included as part of the KDM. The advantages and disadvantages of HW and LW extensions were already stated many times in literature, but basically, the main are that LW extensions are more easily to be incorporated and used in tools, but they provide less precise semantics. On the other hand, HW extensions are more difficult to be incorporated in tools, but they are much more precise in semantics.

Although this paper has concentrated on aspect-oriented extensions, the process we have used for creating the extensions can be generalized, as well as most findings. For example, one of the generalizable findings is the perception that the creation of KDM extensions for a specific domain can be based on existing UML profiles, as we did with Everman’s profile. As the Code Package of KDM has many similarities with UML, it is quite simple to find out which KDM metaclass can be used as base metaclass in a new extension. To assist software engineers in this task with developed a mapping table (UML – KDM) shown in Table 3.

The choice of which KDM extension to use (HW or LW) is guided by several points. Usually, it depends on the goal of the projects. For instance, if the HW mechanism is chosen, the new metaclasses could be instantiated more easily and this provides a better correcteness in instance level, but it makes difficult the interoperability with other tools that uses KDM. On the other hand, the LW extensions is more interoperable but harder to be instantiated and the correctness of the models should be granted by the supported tools that implement it.

As a main limitation we claim that the ADM approach do not have a wide set of tools that works with KDM to facilitate the reverse and forward engineering available in the literature. Thus, we are depending on specific tools and programming languages, such as MoDisco tool and JAVA programming language. However, we be-

lieve that the guidelines provided in this paper could be used in another programming language because we explain how to use the KDM to perform the extensions.

The usage scenario of KDM extensions as well as the right moment to use them are still not so clear. Along aspect-oriented modernization projects, instances of the extended KDM will be created as a result of refactorings/optimizations/reestructuring tasks applied over the legacy KDM (the KDM that represents the legacy system). This happens in the upper right part of the horse shoe model. We believe that, in many situations, the instances of the new concepts (metaclasses or stereotypes) of the extended KDM will be automatically created by the transformation rules. For example, a transformation rule could get as input a legacy KDM with some packages annotated with crosscutting concerns and generate aspects for each annotated package. However, there are many other situations where the software engineers will have to create the transformations manually for instantiating the new aspect-oriented concepts. In these cases, it is important to know the differences between HW and LW extensions and the advantages and disadvantages of them.

We are currently studying how KDM and KDM extensions can be arranged into the architecture of modernization tools. In this sense, we are developing a Reference Architecture for supporting the design of this kind of tools (Santos and de Camargo, 2016). As part of this effort, we are also working on creating a terminology that better characterize these tools.

The experimental analysis presented in this paper has considered the development and maintenance on KDM instances. These activities were performed in Eclipse IDE without the help of a tool to simplify the codification process. Thus, as a future work we envision the opportunity of developing a modernization tool that allows the automatic instantiation of KDM-AO elements (HW and LW) to easier the creation of aspect-oriented refactorings using KDM. With this modernization tool would be possible to reapply the experiment with a bigger set of subjects in order to evaluate other quality attributes such as usability and quality of KDM-AO instances in real company projects. Thus, with this experiment we could be able to find out if the modernization process proposed by ADM is suitable in the context of companies.

As another future work, we intend to conduct other case studies using other aspect-oriented languages such as AspectC++ and AspectS in order to evaluate if our KDM-AO extensions are generic and platform independent enough to represent them.

Acknowledgements

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) Finance Code 001 (grant number 88881.131912/2016/0).

Daniel San Martin would like to thank CONICYT (Chile) (grant number 72170024). André Landi would like to thank S2IT SOLUTIONS CONSULTORIA LTDA. Valter Camargo would like to thank FAPESP (process number 2016/03104-0).

References

Ahmed, R.A.M., Aboutabl, A.E., Mostafa, M.-S.M., 2017. Extending unified modeling language to support aspect-oriented software development. Int. J. Adv. Comput. Sci. Appl. 8 (1), 208–215.

Baresi, L., Miraz, M., 2011. A component-oriented metamodel for the modernization of software applications. In: 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, pp. 179–187. doi:10.1109/ICECCS.2011.25.

Bates, D., Mächler, M., Bolker, B., Walker, S., 2015. Fitting linear mixed-effects models using lme4. J. Stat. Softw. 67 (1), 1–48. doi:10.18637/jss.v067.i01.

Bianchi, A., Caivano, D., Marengo, V., Visaggio, G., 2003. Iterative reengineering of legacy systems. IEEE Trans. Softw. Eng. 29 (3), 225–241. doi:10.1109/TSE.2003.1183932.

Bruneliere, H., Cabot, J., Jouault, F., Madiot, F., 2010. Modisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA, pp. 173–174. doi:10.1145/1858996.1859032.

de Camargo, V.V., Masiero, P.C., 2008. An approach to design crosscutting framework families. In: Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software. ACM, New York, NY, USA, pp. 3:1–3:6. doi:10.1145/1404891.1404894.

Chagas, F., Durelli, R., Terra, R., Camargo, V., 2016. KDM as the underlying metamodel in architecture-conformance checking. In: Proceedings of the 30th Brazilian Symposium on Software Engineering. ACM, pp. 103–112.

Chen, L., Wang, J., Xu, M., Zeng, Z., 2010. Reengineering of java legacy system based on aspect-oriented programming. In: 2010 Second International Workshop on Education Technology and Computer Science, vol. 3, pp. 220–223. doi:10.1109/ETCS.2010.298.

Clausen, A., 2012. Transforming Python into KDM to Support Cloud Conformance Checking. Ph.D. thesis. Kiel University.

Cook, T.D., Campbell, D.T., 1979. Quasi-Experimentation: Design & Analysis Issues for Field Settings. Houghton Mifflin.

Durelli, R.S., Santibáñez, D.S.M., Delamaro, M.E., de Camargo, V.V., 2014a. Towards a refactoring catalogue for knowledge discovery metamodel. In: Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014), pp. 569–576. doi:10.1109/IRI.2014.7051940.

Durelli, R.S., Santibáñez, D.S.M., Marinho, B., Honda, R., Delamaro, M.E., Anquetil, N., de Camargo, V.V., 2014b. A mapping study on architecture-driven modernization. In: Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014), pp. 577–584. doi:10.1109/IRI.2014.7051941.

Durelli, R.S., Viana, M.C., de S. Landi, A., Durelli, V.H.S., Delamaro, M.E., de Camargo, V.V., 2017. Improving the structure of KDM instances via refactorings: an experimental study using KDM-re. In: Proceedings of the 31st Brazilian Symposium on Software Engineering. ACM, New York, NY, USA, pp. 174–183.

Evermann, J., 2007. A meta-level specification and profile for AspectJ in UML. In: Proceedings of the 10th International Workshop on Aspect-oriented Modeling. ACM, New York, NY, USA, pp. 21–27. doi:10.1145/1229375.1229379.

Hohenstein, U.D., Jäger, M.C., 2009. Using aspect-orientation in industrial projects: appreciated or damned? In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, pp. 213–222. New York, NY, USA.

Júnior, J.U., Penteado, R.D., de Camargo, V.V., 2010. An overview and an empirical evaluation of UML-AOF: an UML profile for aspect-oriented frameworks. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 2289–2296. doi:10.1145/1774088.1774564.

Kazman, R., Woods, S.G., Carrière, S.J., 1998. Requirements for integrating software architecture and reengineering models: Corum II. In: Proceedings of the Working Conference on Reverse Engineering (WCRE'98). IEEE Computer Society, Washington, DC, USA, pp. 154–160.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: ECOOP'97–Object-Oriented Programming, pp. 220–242.

Kulesza, U., Soares, S., Chavez, C., Castor, F., Borba, P., Lucena, C., Masiero, P., Sant'Anna, C., Ferrari, F., Alves, V., Coelho, R., Figueiredo, E., Pires, P.F., Delicato, F., Piveta, E., Silva, C., Camargo, V., Braga, R., Leite, J., Lemos, O., Mendonça, N., Batista, T., Bonifácio, R., Cacho, N., Silva, L., von Staa, A., Silveira, F., Valente, M.T., Alencar, F., Castro, J., Ramos, R., Penteado, R., Rubira, C., 2013. The crosscutting impact of the AOSD Brazilian research community. J. Syst. Softw. 86 (4), 905–933. SI : Software Engineering in Brazil: Retrospective and Prospective Views URL http://www.sciencedirect.com/science/article/pii/S0164121212002427. doi: https://doi.org/10.1016/j.jss.2012.08.031.

Landi, A., Chagas, F., Santos, B.M., Costa, R.S., Durelli, R., Terra, R., de Camargo, V.V., 2017. Supporting the specification and serialization of planned architectures in architecture-driven modernization context. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), pp. 327–336.

Lehman, M.M., 1996. Laws of software evolution revisited. In: Montangero, C. (Ed.), Software Process Technology. Springer, Berlin, Heidelberg, pp. 108–124.

Lesiecki, N., 2006. Applying AspectJ to J2EE application development. IEEE Softw. 23 (1), 24–32.

Martín Santibáñez, D.S., Durelli, R.S., de Camargo, V.V., 2015. A combined approach for concern identification in KDM models. J. Braz. Comput. Soc. 21 (1), 10. doi:10.1186/s13173-015-0030-3.

Normantas, K., Sosunovas, S., Vasilecas, O., 2012. An overview of the knowledge discovery meta-model. In: Proceedings of the 13th International Conference on Computer Systems and Technologies. ACM, New York, NY, USA, pp. 52–57. doi:10.1145/2383276.2383286.

OMG, 2009. Architecture-Driven Modernization Standards Roadmap doi:10.1016/j.gie.2008.12.063. Avaliable at http://adm.omg.org/.

OMG, 2016. OMG ® Specifications Business Modeling Specifications. Available at http://www.omg.org/spec/.

Pérez-Castillo, I. G.-R., Piattini, M., 2011. Modern Software Engineering Concepts and Practices: Advanced Approaches, Architecture-Driven Modernization. Chapter 475–103. doi:10.4018/978-1-60960-215-4.

Pérez-Castillo, R., de Guzmán, I.G.-R., Piattini, M., 2011. Knowledge discovery metamodel-ISO/IEC 19506: a standard to modernize legacy systems. Comput. Stand. Interfaces 33 (6), 519–532. doi:10.1016/j.csi.2011.02.007.

Pérez-Castillo, R., de Guzmán, I.G.-R., Piattini, M., Weber, B., 2012. Integrating event logs into KDM repositories. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1095–1102.

Qaisar, Z.H., Anwar, N., Rehman, S.U., 2013. Using UML behavioral model to support aspect oriented model. J. Softw. Eng. Appl. 6 (03), 98.
Rausch, A., Rumpe, B., Hoogendoorn, L., 2003. Aspect-oriented framework modeling. In: Proceedings of the 4th AOSD Modeling with UML Workshop (UML Conference 2003).
Sadovykh, A., Vigier, L., Hoffmann, A., Grossmann, J., Ritter, T., Gomez, E., Estekhin, O., 2009. Architecture driven modernization in practice 150; study results. In: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, pp. 50–57. doi:10.1109/ICECCS.2009.39.
Santos, B.M., de Camargo, V.V., 2016. A reference architecture for KDM-based modernization tools. In: Proceedings of VI Workshop de Teses e Dissertações do CBSOFT (WTDSOFT 2016), pp. 1–9.
Santos, B.M., Durelli, R.S., Honda, R.R., Camargo, V.V., 2014a. Investigating lightweight and heavyweight KDM extensions for aspect-oriented modernization. In: 11th Workshop on Software Modularity (WMod), Maceió, Brazil, pp. 1–12.
Santos, B.M., Honda, R.R., Durelli, R.S., d. Camargo, V.V., 2014b. KDM-AO: an aspect-oriented extension of the knowledge discovery metamodel. In: 2014 Brazilian Symposium on Software Engineering, pp. 61–70. doi:10.1109/SBES.2014.20.
Schutter, K.D., Adams, B., 2007. Aspect-orientation for revitalising legacy business software. Electron Notes Theor. Comput. Sci. 166, 63–80. Proceedings of the ERCIM Working Group on Software Evolution (2006). doi: https://doi.org/10.1016/j.entcs.2006.08.002URL http://www.sciencedirect.com/science/article/pii/S1571066106005299.
Shahshahani, P.M., 2011. Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming. Concordia University Master's thesis. URL http://spectrum.library.concordia.ca/7329/.
Soares, S., Laureano, E., Borba, P., 2002. Implementing distribution and persistence aspects with AspectJ. SIGPLAN Not. 37 (11), 174–190. doi:10.1145/583854.582437.
Stein, D., Hanenberg, S., Unland, R., 2002. A UML-based aspect-oriented design notation for AspectJ. In: Proceedings of the 1st international conference on Aspect-oriented software development. ACM, pp. 106–112.
Ulrich, W.M., Newcomb, P., 2010. Information Systems Transformation: Architecture-Driven Modernization Case Studies. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
Visaggio, G., 2001. Ageing of a data-intensive legacy system:symptoms and remedies. J. Softw. Mainten. 13 (5), 281–308. URL http://dl.acm.org/citation.cfm?id=565153.565154.
Wobbrock, J.O., Findlater, L., Gergle, D., Higgins, J.J., 2011. The aligned rank transform for nonparametric factorial analyses using only Anova procedures. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, pp. 143–146. doi:10.1145/1978942.1978963.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA.
Zakaria, A.A., Hosny, H., Zeid, A., 2002. A UML extension for modeling aspect-oriented systems. In: International Workshop on Aspect-Oriented Modeling with UML, Germany.

Bruno Marinho Santos is graduated in information systems at Faculdade Integral Diferencial (FACID) in 2010 and he obtained his master degree in computer science in software engineering area at Federal University of São Carlos (UFSCar) in 2014. Nowadays, he is a Ph.D. student at UFSCar. He has experience in computer science area, with emphasis in Computation Systems, acting mainly in the following subjects: Aspect-Oriented Modernization, Architecture-Driven Modernization, Crosscutting Frameworks, Knowledge Discovery Metamodel, and metamodel extensions.

André de Souza Landi is a system analyst at S2IT SOLUTIONS CONSULTORIA LTDA working in a national project of UOL. He finished his master's at University of São Carlos - UFSCar/DC in 2018. Nowdays, he is researching about the topics of Software Architecture, Modularity, Model-Driven Engineering and new techniques and framework for the Java language.

Daniel San Martín was chief information security officer in the Information Technology Department at Universidad Austral, Valdivia, Chile until February 2016. Prior to joining the IT department, he was project manager and Information Analyst in several public and private business like Mining Industry, Educational Institutions and IT Consultory. He received a B.S. degree in Engineering Science and Computer Engineering from Universidad Católica del Norte, Antofagasta, Chile and M.Sc. degree in computer science from Universidade Federal de São Carlos, SP, Brazil. Currently, he is a Ph.D. Student at Universidade Federal de São Carlos, Brazil. His research interests in computer science are in the area of software engineering with special interest in adaptive systems.

Rafael S. Durelli is professor at Science Computer Department of Federal University of Lavras (UFLA) in Brazil. He finished his Ph.D. at University of São Paulo - USP/ICMC in 2016. He is a member of PqES/DCC (Pesquisa em Engenharia de Software).

Valter Vieira de Camargo is an associate professor at Computing Department of the Federal University of São Carlos (UFSCar) in Brazil. He has co-authored around 130 research papers, covering the topics of Software Architecture, Software Modernization, Adaptive Systems, Modularity and Model-Driven Engineering. He finished his Ph.D. in 2006 and participated as a visiting researcher at University of Twente in 2013. He has also coordinated the AdvanSE (Advanced Research on Software Enginering) Group since 2009.