Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization

Bruno M. Santos[1], Rafael S. Durelli[2], Raphael R. Honda[1], Valter V. Camargo[1]

[1]Departamento de Computação Universidade Federal de São Carlos - UFSCar
São Carlos – SP – Brazil.

[2]Instituto de Ciências Matemáticas e Computação Universidade de São Paulo - USP
São Carlos – SP – Brazil.

{bruno.santos, raphael.honda, valter}@dc.ufscar.br,
rsdurelli@icmc.usp.br

Abstract. *Architecture-Driven Modernization is the new generation of software reengineering. The main idea is to modernize legacy systems using a set of standard metamodels. The first step is to obtain an instance of an ISO metamodel, called KDM, through reverse engineering. Then, refactorings and optimizations can be performed over this model transforming it into a target KDM. Further, a modern source code can be generated from the target KDM. Although KDM intentionally does not provide support for Aspect-Oriented Programming (AOP), it is possible to extend it by either creating/changing its existing metaclasses (heavyweight) or using its extension mechanism, creating stereotypes (lightweight). There are some few works in the literature that present KDM extensions. However, all of them present just one extension without providing a discussion or a comparison about the suitability of them. This is important because such comparison can support modernizers in choosing the best alternatives to meet their needs. Therefore, in this paper we present two KDM extensions for AOP in order to enable an AO modernization. We also present a preliminary comparison between them trying to characterize their vantages and disadvantages and we conclude that to perform an AO modernization the heavyweight one is the most indicated.*

1. Introduction

Architecture Driven Modernization (ADM) was proposed by Object Management Group (OMG) in 2003. ADM uses the concepts of Software Reengineering (SR) and the principles of Model-Driven Architecture (MDA) in order to perform modernizations in a respective system based on models [OMG 2014]. ADM introduces several standards metamodel mainly to support the whole process of model-driven modernization. Its main metamodel is Knowledge Discovery Metamodel (KDM), which is language and platform independent and allows representing various artifacts of a system at different abstraction levels, such as configuration files, database, source-code, etc [ADM 2014].

In a parallel research line, researchers have been using Aspect-Oriented Programming (AOP) as an alternative for the modularization of systems [Kiczales, *et al.* 1997]. They argue that modularize the system is an important kind of modernization because it allows, for instance, the separation of crosscutting concerns in the legacy

source code [Laddad 2003]. In this context, our main goal is to enable the Aspect-Oriented (AO) modernization to be carried out according to the ADM standards. To fulfill this goal an extension of AOP concepts has to be performed in KDM metamodel.

The latest version of KDM is 1.3 [KDM 2012], in this version it is not possible to represent the concepts of AOP. Therefore, it is necessary to extend the metamodel to allow the representation of AOP. Similarly, as UML, nowadays there are two ways to extend the KDM metamodel, either by (*i*) performing a lightweight (LW) extension or (*ii*) performing a heavyweight (HW) extension. Studies published in literature [Mirshams 2011] [Baresi and Mirazi 2011] have performed extensions in KDM, but, most of them do not report the advantages and disadvantages of each extension mechanism.

An important point to perform metamodel extensions is to use a representation of the new concepts that should be represented. In the lightweight extension this representation is called as profile. To support our extensions, we carried out a search in the literature to find a profile that was as comprehensive as possible and which would be able to represent not only the concepts of AspectJ language, but also the concepts of other languages that implement AOP.

One of the most important steps in a modernization process is the refactoring of the models. In this sense, modernization engineers usually need to write source code to refactoring legacy KDM into target KDM. When the modernization is performed in a specific domain, such as AOP, the refactoring code involves to instantiate an extended KDM, which can be a LW or HW extension. Depending on the extension type (light or heavy) the productivity of the modernization engineers can be different because these two types of extensions encompass many particularities [Magableh, Shukur and Ali 2012]. In this paper, we perform these two extensions both named KDM-AO. In other words we have created both LW and HW. Moreover, we present a preliminary comparison between the two extensions we have developed. This comparison aims at providing a base for modernization engineers in choosing the most suitable one according to their needs. The comparison was performed based on some criteria we have chosen and applied to a case study. Our results show that the heavyweight extension is the most appropriate mechanism to deal with domain-specific concepts with a large number of concepts because it helps the productivity and ensure the quality of the models. Also a fully list that contains the advantages and disadvantages of each extension was created. We claim that by means of it the engineer can pinpoint which extension is better for she/he context. In order to assess our KDM-AO we carried out a Crosscutting Framework-based modernization process in a management system of a CD Shop [Camargo and Masiero 2005] using the two extensions. The evaluation showed that both extensions are able to represent all the details inherent in this type of framework, as well as all AO concepts. In addition, the results show that by using the KDM extensions is possible to modernize a legacy system to AOP.

This paper is structured as follows: In Section II, the background about ADM/KDM is shown. In Section III the Metamodel Extensions are described. A case study is shown in Section IV. In Section V we present a preliminary qualitative comparison. The related works are shown in Sections VI. Finally, in Section VII, the discussions and conclusions are presented.

2. Background

In 2003, OMG initiated efforts to standardize the process of modernization of legacy systems using models by means of the ADM Task Force (ADMTF) [ADM 2014]. The aim of the ADM is the revitalization of existing applications by adding or improving functionalities, using the OMG modeling standards and also considering MDA principles. In order to support the modernization process, in 2006 the KDM metamodel was created. KDM is language and platform-independent.

KDM contains twelve packages and it is structured in a hierarchy of four layers: (i) Infrastructure Layer, (ii) Program Elements Layer, (iii) Runtime Resource Layer and (iv) Abstractions Layer [KDM 2012]. These layers are created automatically, semi-automatically or manually through the application of various techniques of extraction of knowledge, analysis and transformations [5]. Each layer is organized into packages that define a set of metamodel, whose purpose is to represent a specific and independent interest of knowledge related to legacy systems [KDM 2012]. In the context of this paper, we are going to focus on the Infrastructure Layer and Program Elements Layer, especially on the *KDM* and *Code* packages. The KDM package has a set of metaclasses that allows the creation of stereotypes in instantiation time. This means that KDM package provides the metaclasses to perform a LW extension, so it is not needed the creation of new metaclasses in KDM metamodel.

The *Code* package contains a set of metaclasses to represent program elements in implementation level. This package owns a set of metaclasses that represent the common named elements in the source code supported by different programming languages such as data types, classes, procedures, methods and interfaces [KDM 2012].

The KDM metamodel provides the base metaclasses that allows performing both light and heavyweight extensions, the decision of which kind of extension will depend on the modernization context. When the LW extension is performed, the original metamodel is not changed, so this guarantees the tools compatibility. Otherwise, the HW extension changes the original metamodel adding new metaclasses, so new tools have to be made to support this kind of extension. These points are valid for LW and HW extension when performed in KDM or Unified Modeling Language (UML) metamodels, even the goal of these two metamodels being different [Pérez-Castillo, Guzmán and Piattini 2011].

Another important point is that KDM is not a metamodel intended to serve as base for diagrams, like UML [Chavez and Lucena 2002] [Fuentes and Sanchez 2006]. While humans usually create UML instances, KDM instances are created by parsers and processed by tools. Therefore, profiles make more sense in the context of UML than in the context of KDM. Anyway, this paper aims to perform both extensions in KDM metamodel and list the advantages and disadvantages, providing guidance on how to perform them.

3. Metamodel Extensions

As in UML, it is also possible to define either LW or HW extension in KDM by means of extension mechanism. HW extensions are based on a modified KDM metamodel, including new metaclasses or changing the existing ones. In other hand, LW extensions (profiles) are based on set of *Stereotypes* and *TagDefinitions*, which are basically

"notes" over the model. Profiles are able to impose restrictions on existing metaclasses, but they respect the metamodel, without modifying the original semantics of the elements. One of major benefits of profiles is that they can be handled in easier way by existing tools.

In general, the drawback of HW extensions is that existing tools get no longer compatible with the new metamodel. However, the only way to guarantee model correctness in model level is using HW extensions. This happens because it is possible to relate metamodel elements by their types and not just by their names, as it usually happens in LW extensions. Using LW extensions, the correctness of the model must be guaranteed by tools.

The main decision before performing KDM extension was to choose an UML profile which was broad enough to represent all the AO concepts. In this sense, we conducted a literature review to identify Aspect-Oriented metamodels and UML profiles that could be considered good candidates. We had analyzed several proposals, but the Evermann's profile was considered the most suitable one because it incorporates the level of details that we are interested in **Erro! Fonte de referência não encontrada.**. Both extensions presented in this paper have been created based on Evermann's profile. In Figure 1 is presented both extensions. We would like to emphasize that a more detailed version of the HW extension can be found in **Erro! Fonte de referência não encontrada.**. Each class/element has four words (four lines) in its first compartment. The first word (in bold) represents the name of the metaclass we have created in our heavyweight extension, for example, *AspectUnit* is a term that exists in the HW extension. The second word, inside the brackets, is the KDM metaclass we have chosen to make the current element extends from. For example, the new metaclass *AspectUnit* that belongs to our heavyweight extension extends the *ClassUnit* KDM metaclass.

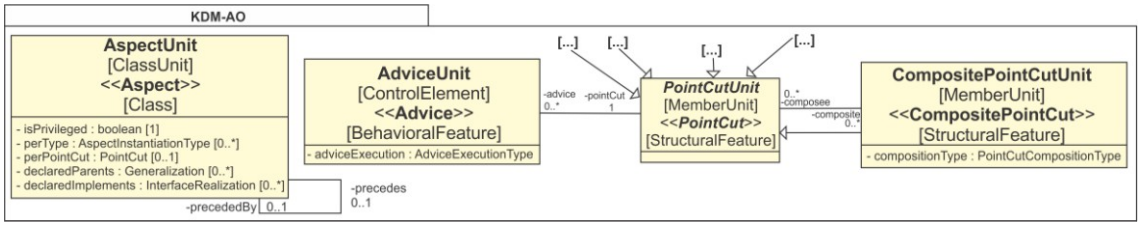

Figure 1. KDM-AO and Evermann's Profile (Adapted)

In our lightweight extension, we do not have metaclasses, but stereotypes. Thus, below the first two elements already described there is a <<stereotype>>. Therefore, all the stereotypes shown in this figure exist in our lightweight extension. Stereotypes are applied over existing elements, so the name of the element in which the stereotype can be applied on is the metaclass shown in the second line. For example, the stereotype <<aspect>> can only be applied over *ClassUnit* instances. The last line contains the name of the UML metaclass used in the original version of Everman's profile.

3.1. Lightweight extension

KDM metamodel provides a set of metaclasses in *KDM* package to allow the creation of stereotype families; these metaclasses are shown in Figure 2. The *ExtensionFamily* element acts as a container for a set of related stereotypes and their corresponding *TagDefinitions*. The *Stereotype* concept provides a way of branding model elements so

that they behave as if they were the instances of new virtual metamodel constructs. Thus, a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure.

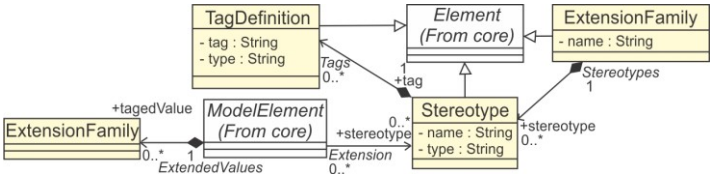

Figure 2. Extensions Class Diagram (Adapted)

Each *Stereotype* owns the optional set of *TagDefinitions* and each one provides the name of the tag and the name of the KDM type of the corresponding value. The *ExtensionFamily*, *Stereotype* and *TagDefinition* elements are the main elements to represent lightweight extensions in KDM. Herein, we are representing the LW extension programmatically. Therefore, to perform this extension the user has to create a new Java class and create instances of the *ExtensionFamily*, *Stereotype* and *TagDefinition* elements with the values that represent the new elements. In Figure 3 is represented a snippet of the programmatic lightweight extension in KDM. Line 1 depicts the creation of *ExtensionFamily's* element instance and Line 2 shows an instance of *Stereotype* element. Line 3 adds the *Stereotype* created on Line 2 into the *ExtensionFamily* created on Line 1. Lines 4 and 5 the values of *Stereotype's Name* and *Type* are set. In the Line 6 a *TagDefnition* is created and in the Line 7 this tag is attached to the *Stereotype*. Finally, in the Lines 8 and 9 the *TagDefinition's* properties *Tag* and *Type* are defined.

```
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"); [...]
```

Figure 3. Snippet of LW extension programmatically

Once all the elements in Figure 1 have been added programmatically it is possible reuse it by means of a class with all the *Stereotypes* and *TagDefinitions* programmed. The usage of light and heavyweight extension is shown in Section 4. In Section 4 we present a modernization case study that uses both extensions to make the representation of the main AOP concepts.

3.2. Heavyweight extension

In Evermann's profile, the *CrossCuttingConcern* element extends the Package UML metaclass and aims to represent the existence of a Crosscutting Concern like Persistence, Security and Concurrency. In our KDM-AO this element extends the Package metaclass. This KDM metaclass represents a package in which is possible to deposit Aspects, Classes and other elements.

AspectUnit is our element for representing aspects, which extends the *ClassUnit* KDM metaclass. We decided to extend this metaclass because aspects have all the characteristics that classes have, but pointcuts, advices and intertype declarations. The element for representing advices in our heavyweight extension is *AdviceUnit*, which

extends the *ControlElement* metaclass. Knowing that advice is an element that specifies behavior, we could consider it like a method. However, advices do not have neither access specifiers (public, private, protected) nor types (constructor, destructor, etc). Because of that we have decided do not make it extends *MethodUnit*, instead, we make it extends from its parent metaclass *ControlElement*.

PointCutUnit is our element for representing pointcuts. According to Evermann's profile, pointcut is a structural element and extends the UML metaclass *StructuralFeature*. KDM has also a class for representing structural characteristics called *DataElement*, which is an abstract metaclass. Its descendents are *StorableUnit*, *MemberUnit*, and *ItemUnit*. As *StorableUnit* and *ItemUnit* cannot be *abstract*, *MemberUnit* was chosen to be the supertype of *PointCutUnit*. Moreover, another reason for extending *MemberUnit* was that pointcuts can crosscut other classes and *MemberUnit* is the KDM metaclass used to denote references to other classes. *StaticCrossCuttingFeature* is our element for representing intertype declarations. In our KDM-AO we have decided to extend two KDM metaclasses: *StorableUnit* e *MethodUnit*. In this way, *StaticCrossCuttingFeature* is able to represent structural and behavioral characteristics.

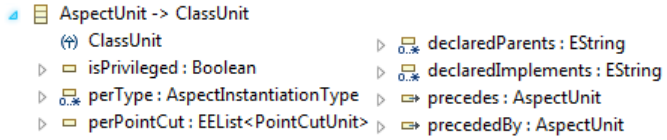

Figure 4. A snippet of the heavyweight extension

In Figure 4 is possible to see the *AspectUnit* element expanded with all its attributes and relationships. As we can see, *ClassUnit* is the superclass of *AspectUnit* and all the attributes and relationship added are present in our KDM-AO.

4. Case Study Overview

In this section is presented a case study in which were applied both extensions in a modernization process based on Crosscutting Frameworks (CFs) **Erro! Fonte de referência não encontrada.**. CFs are AO Frameworks that encapsulate in a generic way just one crosscutting concern, like Persistence, and Security **Erro! Fonte de referência não encontrada.**. In this case study, we modernized a management system of a CD shop. The modernization goal was to modularize the persistence concern with aspects. As our group has some experience with Crosscutting Frameworks, the idea was to use a Persistence CF previously developed in this process. To represent this application using our KDM extensions we have to create 4 Aspects and 2 Pointcuts in both extensions.

Because of space limitations, in this paper we show the usage of our extensions to represent just two aspects, one is from the Persistence CF and the other is from the instantiation model. The name of the first chosen aspect is *ConnectionComposition*. Its purpose is to provide a base behavior for opening and closing database connections. During instantiation, application engineers need to provide concrete implementations for the abstract pointcuts *openConnection*() and *closeConnection*(). This aspect has in its body an attribute, two abstract pointcuts, a concrete and one abstract operation and two advices. In Figure 5 there are parts A and B in which part A represents an instance of LW extension and part B represents an instance of HW extension. Each line contains the

element type and then its value for both extensions. The following paragraphs describe only the AOP elements, the elements from Object-Orientation are out of our scope.

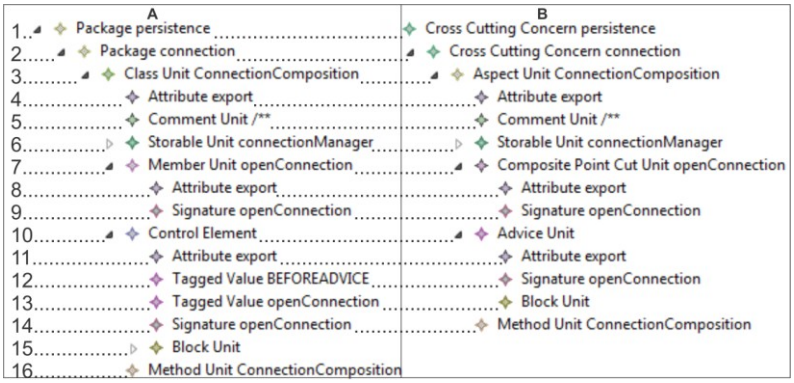

Figure 5. A snippet of the aspect *ConnectionComposition*.aj in KDM-AO LW and HW

In Line 1 we can visualize a Package (Part A) and *CrossCuttingConcern* (Part B) whose value is persistence, i.e., in LW this is an instance of the package metaclass and in HW an instance of *CrossCuttingConcern* metaclass. The main difference between them is that besides they are from different metaclasses, the LW one is stereotyped with *CrossCuttingConcern*. Line 3 displays the name of the aspect that is being modeled here; in LW the stereotype *AspectUnit* is applied in an instance of *ClassUnit* and in the HW there is no stereotype, instead of this we have an instance of the metaclass *AspectUnit*. To declare a Pointcut in LW you have to apply a stereotype in the *MemberUnit* element, and this is shown in Line 7 part A. *CompositePointCutUnit* (Line 7 part B) is used to represent concrete or abstract pointcuts of an aspect.

The element in KDM that can represent an Advice in LW is ControlElement when the stereotype *AdviceUnit* is applied (Line 10). In HW *AdviceUnit* (Line 10) represents the advice that was declared in the aspect. It is essential to fill the Advice Execution property because this property declares what kind of advice that element represents (After, Before or Around). This kind of declaration is very different in both extensions, for example, while in HW this information is set in the properties of the *AdviceUnit* element, in LW you have to create a *TaggedValue* element and apply a TagDefinition in it. In Lines 12 and 13 (part A) we have two *TaggedValues* elements, the first one store the *TagDefinition* adviceExecution and the second one stores the name of the poincut that this *AdviceUnit* (*ControlElement*) belongs.

The second Aspect that has been chosen is from the instantiation model and its name is *myConnectionCompositionRules*. This aspect stores the name of the method in the base source code that will be affected by the Persistence CF and it is composed by two Pointcuts and one method that store the name of the class that will implement the persistence concern. In Figure 6 is depicted a snippet of the aspect *myConnectionCompositionRules* in heavyweight extension. This figure also shows an *ExecutionPointCutUnit*. The *ExecutionPointCutUnit* intercepts the execution of the method main in the class *FindSomeCDs* and is bounded to the *CompositePointCut openConnection*. Line 1 represents the source code to create an instance of *ExecutionPointCutUnit*, according to the heavyweight extension. Lines 2 and 3 represent the declaration of some properties of the *ExecutionPointCutUnit* element, in Line 2 the *operation* is set and in Line 3 is set the *CompositePointCutUnit* that the Execution belongs.

```
1..ExecutionPointCutUnit myExecution =
CodeFactory.eINSTANCE.createExecutionPointCutUnit();
2..myExecution.getOperation().add(FindSomeCDs.main);
3..myExecution.getComposite().add(openConnection); [...]
```

Figure 6. Snippet of *myConnectionCompositionRules* in heavyweight

Figure 7 shows a LW instance of KDM that represents the same source code in Figure 6 but now we are using only metaclasses from the original KDM.

```
1..MemberUnit myExecution = CodeFactory.eINSTANCE.createMemberUnit();
2..myExecution.getStereotype().add(Profile.executionPointCutUnit);
3..TaggedValue operation = KdmFactory.eINSTANCE.createTaggedValue();
4..operation.setTag(Profile.operation);
5..operation.setValue("FindSomeCDs.main");
6..TaggedValue composite = KdmFactory.eINSTANCE.createTaggedValue();
7..composite.setTag(Profiles.composite);
8..composite.setValue("openConnection");
9..myExecution.getTaggedValue().add(operation);
10..myExecution.getTaggedValue().add(composite); [...]
```

Figure 7. Snippet of *myConnectionCompositionRules* in lightweight

To create an *ExecutionPointCutUnit* we have to apply the *execution* stereotype in a *MemberUnit* element, this is represented in Lines 1 and 2. There are two main properties in an *ExecutionPointCutUnit* that are the *operation* and the *composite*, but these properties do not exist in *MemberUnit* element. To declare additional property in an element it is necessary to create instances of *TaggedValue* element and apply a *TagDefinition*. After this it is time to set the value of the property and this value is always a *String Type*. The *operation* property is declared in Lines 3, 4 and 5 and the *composite* property is declared in Lines 6, 7 and 8. Once we have created a *TaggedValue* and set its properties we have to bind the *TaggedValue* to its respective element and the source code that implements this is declared in Lines 9 and 10.

5. A Preliminary Qualitative Comparison

In order to compare the two extensions mechanisms we defined some evaluation criteria shown in Table I. These criteria were created to compare the extensions, trying to characterize if one is better than another one.

The first criterion is intuitiveness of creating instances. This criterion tries to compare the intuitiveness of instantiating the extensions. In this sense, intuitiveness is related to how straightforward is to use the extension elements. For example, to create an aspect in the heavyweight extension it is only necessary to create an instance of the *AspectUnit* class. However, in the lightweight version, it is necessary to create an instance of the *ClassUnit* class and apply the stereotype *AspectUnit* on it. In this case it is much more intuitive to use the HW than the LW one. Notice that this criterion impacts both the creation of new instances of the extensions and also the maintainability of those instances. As HW is more intuitive, it is more straightforward of creating and also changing (adding, removing) existing extension instances. This can be seen if we compare Figure 6 and Figure 7.

The second criterion is the correctness of creating instances. This criterion is related to the likelihood of creating erroneous instances. The heavyweight is better than the light because the values passed in the lightweight extension are only String types, so if the programmer set wrong information this will not be validated. Otherwise, heavyweight extensions are strongly typed, so if a property receives a Boolean type, the

only values possible to set are "*true*" or "*false*", anything else will be considered as a type error and the model will not be able to compile.

The third criterion is tooling impact, which is regarding how easy is to reuse the extension in different tools. The lightweight extension is easier to reuse because tools may be previously designed for considering profiles. In this case, the task would only to import the profile into the tool and use it. However, heavyweight extensions are usually proprietary solutions. Most of the times they are not designed for being reused in other contexts. Although it is not impossible to make a heavy extension reusable, it is harder when compared to the lightweight version. For example, if a tool has been designed for mining KDM instances, it may not work properly if a modified KDM is provided. The impact can be minimized when the heavyweight extension has just added new elements in the standard KDM, rather than modified existing ones.

The fourth criterion is the productivity in creating the extensions. This criterion tries to characterize the effort in creating LW and HW extensions. Clearly, the creation of LW ones is much easier than HW. As creating LW you do not need to worry with types because basically every value is set with String Type. The creation of heavyweight ones is much more time consuming because every additional property have to be set in a *TaggedValue* element. One important point to highlight here is that the productivity in creating extensions is not as important as the maintainability of it. This happens because, in general, the extension will be created once and used several times. Therefore, if the creation takes two months or two hours is not that important. Nevertheless, although the creation of heavy extensions is harder, its usage intuitiveness is better, as result, fewer errors are expected in its instances.

The fifth criterion is about the productivity for maintaining the extension. This criterion tries to characterize the effort in adding, removing and changing existing instances elements. As for maintaining the extension, LW showed itself quicker and easier than HW. This came about once that to perform any change it is necessary just to apply the updates and the LW extension is be already to use. Otherwise the HW extension after applying all changes it is necessary to recompile and update the plug-in.

The sixth criterion is the complexity of the XMI. The idea here is to identify which XMI is more complex. Complex XMI is harder to process and will require much more effort in writing the algorithm. Again, HW extension presents a more clear and intuitive XMI. In this extension the algorithm just need to search for the correct term, such as *AspectUnit*. However, in the LW version, the identification of elements required the identification of the metaclass that represents the element, the identification of the *stereotype* and its *tags*. In Table 1 is possible to see a comparison between LW and HW. There is a ↑ when the extension is better and a ↓ when the extension is worse.

Table 1. Comparison between the extension mechanisms

Criteria	Lightweight	Heavyweight
1. Intuitiveness of Creating Instances	↓	↑
2. Instances Correctness	↓	↑
3.Tooling impact	↑	↓
4.Productivity for Creating the Extensions	↑	↓
5. Productivity for Maintaining the Extension	↓	↑
6. Complexity of the XMI	↓	↑

In terms of reusability of the extension, the LW version is a better alternative because tools can be more easily prepared for working with profiles. For example, consider the existence of a tool that applies refactoring over the KDM. This tool can be

easily extended to consider stereotypes. Although it is not impossible to make a HW extension to be reused, the impact in tools will required more effort.

If you are intending to make your extension available to be reused, than the LW one seems to be a better alternative. This happens because you can make available just the *Stereotypes* and *TaggedValues*. If existing tools were already designed for accepting KDM profiles, it is straightforward to reuse it.

If you are intending to use the extension into an organization or proprietary tool, the HW can be a better solution in terms of productivity and quality of the instances. It's important because the quality of the modernized instance will have a direct impact in the source code generated. Based on our experience in performing LW and HW extensions we consider the HW extension the best mechanism to support AOP modernizations because this kind of modernization involves many concepts and the HW one ensures a higher productivity and quality of the models. The set of criteria presented aims to support software engineers in choosing a type of extensions. The final depends on several other details, context and scenarios in which the extension will be used.

6. Related Works

We found some works in the literature that presents KDM extensions, but we are considering only the two most relevant. The first one is proposed by Mirshams [Mirshams 2011] and another one is proposed by Baresi and Miraz (2011). However, as they have developed just one extension, either light or heavy, they do not provide any comparison in order to characterize which one is better.

Mirshams (2011) created a heavyweight KDM extension for AOP. There are two main differences between our works. The first difference is the level of abstraction 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. The second difference is that her work is limited to dynamic crosscutting as there are no elements for representing intertype declarations. Baresi and Miraz (2011) describe another KDM extension. They proposed a HW KDM extension to support Component-Oriented MOdernization (COMO) that is a metamodel with traditional concepts of software architecture, allowing attaching software components in KDM. This paper performs a KDM extension but the main difference is that they don't use AOP concepts and its goal is just to allow KDM to perform COMO modernizations.

There is another related work that compares LW and HW extension but in UML metamodel. Magableh, Shukur and Ali (2012) gather fourteen papers that executed LW or HW and apply some comparison criteria. As we did here, they list some criteria to evaluate the papers. The criteria used by them were used to inspire us to create ours, but we didn't use these criteria because some of them were too much specific for UML or considered if the tool created to support the extension were complete enough. Because of this we had to create a new set of criteria to support our discussions.

7. Discussions and Conclusions

The main points in this paper are on how to perform these extensions and list the advantages and disadvantages of using them. In section III we showed some guidelines

that can help performing these extensions and in section V we present a list with some criteria that can help deciding which extension mechanism a software engineer should use, of course, depending on his necessities and context.

By means of our case study, it is fairly evident that our extensions can represent all elements of AOP. However, as we have not carried out a complete case study to gauge how reliable our extension is to represent aspects concepts in other programming languages, such as AspectC++, we argue that this is a limitation of our extensions. Nevertheless, to mitigate this limitation the elements of AspectC++ and AspectS were analyzed. Consequently, we conclude that there are enough elements in our extension that can be used to represent source code in both AOP languages. The main contribution of the case study is to prove that the AOP concepts can be represented in LW and HW, so we conclude that even if we perform this modernization in another system or use another CF with a different concern we would reach the same results. To prove this we are planning to perform AOP modernizations in other systems with different CFs.

When conducting our case study using CFs, we have noticed that our extension would be more useful and more expressive if it had also metaelements for representing CF characteristics, like hot spots, frozen spots and other framework characteristics. We intend to perform these modifications in a future work [Camargo and Masiero 2008]. As we are proposing a preliminary qualitative comparison our set of criteria may not be enough to compare the extensions. To solve this gap we intend to search in the literature more criteria to ensure a fairer comparison. As other future works, we aim to conduct others case studies using AspectC++ and AspectS in order to test the KDM-AO extensions, to evaluate the issue of platform independence. Another future work that we are already performing is to evaluate these two extension mechanism by means of a controlled experiment in the context of Computer Science graduation students at UFSCar. It consists in dividing the class in two groups and giving them some evolution and maintenance activities to be performed with LW and HW extension. The main point is to evaluate which approach allows the creation and maintenance of model with the less time and lowest number of errors. We intend to publish these results in a paper after we finish all the programmed steps.

By conducting this research we have noticed that the power of model-driven modernization is greatly influenced by the capacity of representing specific concepts in a proper and suitable way. We also noticed that KDM is a powerful metamodel that can be adapted in almost every context, so it will depend on the software engineer to create the solution the best serves to his proposals.

Acknowledgements

Bruno M. Santos and Raphael R. Honda would like to thank CNPq for sponsoring our research. Rafael S. Durelli and Valter V. de Camargo would like to thank the financial support provided by FAPESP, processes numbers 2012/05168-4 and 2014/14080-9, respectively.

References

Architecture-Driven Modernization. (2014) Document omg/ http://adm.omg.org/.

Baresi, L. and Miraz, M. (2011) “A Component-oriented Metamodel for the Modernization of Software Applications”, 16th IEEE International Conference on Engineering of Complex Computer Systems.

Camargo, V. V. and Masiero, P. C. (2005) “Frameworks Orientados a Aspectos”, XIX Simpósio Brasileiro de Engenharia de Software, Uberlândia. pp. 200-216.

Camargo, V. V. and Masiero, P. C. (2008) “An Approach to Design Crosscutting Framework Families”, ACP4IS 08, Brussels, Belgium.

Chavez, C. and Lucena, C. (2002) “A metamodel for aspect-oriented modeling”, In Proceedings of the AOM with UML workshop at AOSD.

Evermann, J. (2007) “An overview and an empirical evaluation of UML: an UML profile for aspect-oriented frameworks”, Workshop AOM ’07, Vancouver, British Columbia, Canada.

Fuentes, L. and Sanchez, P. (2006) “Elaborating UML 2.0 profiles for AO design”, In Proceedings of the AOM workshop at AOSD.

Kiczales, G. *et al.* (1997) “Aspect Oriented Programming”, In: Proceedings of 11 ECOOP. pp. 220-242.

Knowledge Discovery Meta-Model. KDM Guide, April 2012. Document omg/formal/2012-05-08.

Laddad, R. (2003), AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, Greenwich (74° w. long.). pp.75-77.

Magableh, A., Shukur, Z. and Ali, N. (2012) “Heavy-Weight and Light-weight UML Modelling Extensions of Aspect-Orientation in the Early Stage of Software Development”, Journal of Applied Sciences.

Mirshams, P. S. (2011) “Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming”, 79 f. Dissertation (Master of Applied Science in Software Engineering) – University of Montreal, Quebec, Canada, unpublished.

Object Management Group. (2014) OMG Specifications, April 2014. Documents omg/ http://www.omg.org/spec/.

Pérez-Castillo, R., Guzmán, I. G. and Piattini, M. (2011) “Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems”, Computer Standards & Interfaces 33, Elsevier B.V. pp. 519–532.

Santos, B. M., Honda, R. R., Durelli, R. S. and Camargo, V. V. (2014) “KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel”, In 28th Brazilian Symposium on Software Engineering (SBES). Publishing Press.