KDM-RE: A Model-Driven Refactoring Tool for KDM

Rafael S. Durelli[1], **Daniel S. M. Santibáñez**[2] , **Márcio E. Delamaro**[1] **and Valter V. de Camargo**[2]

[1]Computer Systems Department University of São Paulo - ICMC
São Carlos, SP, Brazil.

[2]Computing Departament
Federal University of São Carlos - UFSCAR
São Carlos, SP, Brazil.

{rdurelli, delamaro}@icmc.usp.br[1], valter@dc.ufscar.br[2]

Abstract. *Software refactoring is a proven technique that aims at improving the quality of source code. However, with the advent of Architecture-Driven Modernization (ADM) and its meta-models, it seems a good alternative for shifting from source-code refactoring to model-driven refactoring. The main goal of this shift is to create a language and platform independent catalog of refactoring. Nevertheless, although ADM provides the process for refactoring legacy systems, there is a lack of an Integrated Development Environments (IDEs) to lead engineers to apply refactorings as such exist for other object-oriented languages. We describe a tool, implemented as an Eclipse plug-in designed to fulfill exactly this need. This plug-in supports 17 refactorings that are heavily inspired by refactorings well known in the literature and yet keeping them in synch with the underlying source code as well as a class diagram.*

1. Introduction

Architecture-Driven Modernization (ADM) has been proposed by Object Management Group (OMG) in order to carry out refactoring following the principles of Model-Driven Architecture (MDA) [Kleppe et al., 2003]. It has become a great candidate since it aims to promote *(i)* portability, *(ii)* interoperability, and *(iii)* reusability by means of model. According to the OMG the most important artifact provided by ADM is the Knowledge Discovery Metamodel (KDM). KDM is an OMG specification adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems. KDM is structured in a hierarchy of four layers; *Infrastructure Layer*, *Program Elements Layer*, *Runtime Resource Layer*, and *Abstractions Layer*. We are specially interested in the *Program Elements Layer* because it defines the Code and Action packages which are widely used by our tool. The Code package defines a set meta-classes that represents the common elements in the source-code supported by different programming languages such as: (*i*) `ClassUnit` and `InterfaceUnit` which represent classes and interface, respectively, (*ii*) `StorableUnit` which illustrates attributes and (*iii*) `MethodUnit` to represent methods, etc. The Action package represent behavior descriptions and control-and-data-flow relationships between code elements.

On the other hand, refactoring has been known and highly used both industrially and academically. It is a form of transformation that was initially defined by Opdyke [Opdyke, 1992] as "a change made to the internal structure of the software while preserving its external behavior at the same level of abstraction". Nowadays it is possible to identify several catalogs of refactoring for different languages and the most complete and influential was published by Fowler in [Fowler et al., 2000]. However, while software reengineers would greatly benefit from the possibility to assess different choices during the refactorings, in practice they mostly rely on experience or intuition because of the lack of approaches providing comparison between possible variations of a change. Therefore, software reengineers do not have the possibility to easily apply analyses on different source-code version branches of a system and compare them to pick up the most adequate changes. In this context, the motivations for moving from source-code refactoring to model-driven refactoring are: (*i*) a model provides an abstract view of the system, hence, visualizations of the structural changes required are easier, (*ii*) problems uncovered at the design-level can be improved directly on the model, and (*iii*) using refactoring in high abstract level can allow the software engineer to explore alternate design paths in much cheaper than software refactoring.

To overcome the described problems, we devised a *plug-in* on the top of the Eclipse Platform named **K**nowledge **D**iscovery **M**odel-**R**efactoring **E**nvironment (KDM-RE). This *plug-in* provides an environment for model-driven refactoring based on KDM models, i.e., we provide a *plug-in* with the ability to improve existing KDM models, yet keeping them in synch with the underlying source-code as well as a class diagram. In addition, this *plug-in* supplies a multiple versions of a system at level models (KDM), enabling to the engineer to work interactively on multiple models and to explore alternate refactoring path. Notice that KDM-RE supports 17 refactorings that are heavily inspired by the refactorings described by Fowler [Fowler et al., 2000].

This paper is organized as followed: Section 2 provides the background to fully understand our *plug-in* - Section 3 depicts information upon the *plug-in* KDM-RE and an case study - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. Background

This section introduces the basic concepts of Architecture-Driven Modernization, Knowledge-Discovery Meta-model and refactorings.

2.1. ADM and KDM

OMG defined ADM initiative [Perez-Castillo et al., 2009] which advocates carrying out the reengineering process considering MDA principles. In other words, ADM is the concept of modernizing existing systems with a focus on all aspects of the current systems architecture and the ability to transform current architectures to target architectures by using all principles of MDA [Ulrich and Newcomb, 2010].

To perform a system modernization, ADM introduces Knowledge Discovery meta-model (KDM). KDM is an OMG specification adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems. According to Pérez-Castillo et al., [Perez-Castillo et al., 2009] the goal of the KDM standard is to define a meta-model to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, etc.). The KDM provides a comprehensive high-level view of the behavior,

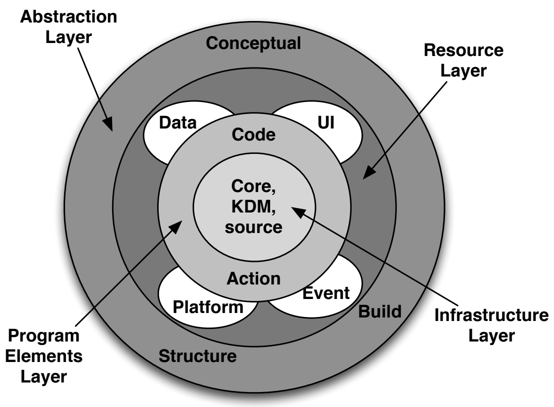


Figure 1. Layers, packages, and separation of concerns in KDM (Adapted from [OMG, 2012])

structure and data of legacy information systems by means of a set of facts. The main purpose of the KDM specification is not the representation of models related strictly to the source code nature such as Unified Modeling Language (UML). While UML can be used to mainly to visualize the system "as-is", an ADM-based process using KDM starts from the different legacy software artifacts and builds higher-abstraction level models in a bottom-up manner through reverse engineering techniques.

As outlined before, the KDM consists of four abstraction layers: (*i*) *Infrastructure Layer*, (*ii*) *Program Elements Layer*, (*iii*) *Runtime Resource Layer*, and (*iv*) *Abstractions Layer*. Each layer is further organized into packages, as can be seen in Figure 1. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing software systems. We are specially interested in the *Program Elements Layer* because it defines the Code and Action packages which are widely used by our catalogue. The Code package defines a set meta-classes that represents the common elements in the source code supported by different programming languages. In Table 1 is depicted some of them. This table identifies KDM meta-classes possessing similar characteristics to the static structure of the source code. Some meta-classes can be direct mapped, such as Class from object-oriented language, which can be easily mapped to the `ClassUnit` meta-class from KDM.

Table 1. Meta-classes for Modeling the Static Structure of the Source-code

Source-Code Element	KDM Meta-Classes
Class	ClassUnit
Interface	InterfaceUnit
Method	MethodUnit
Field	StorableUnit
Local Variable	Member
Parameter	ParameterUnit
Association	KdmRelationShip

2.2. Refactoring and Model-Driven Refactoring

In the area of object-oriented programming, refactorings are the technique of choice for improving the structure of existing code without changing its external behavior [Fowler et al., 2000]. They have proved to be useful to improve the quality attributes of source code, and thus, to increase its maintainability. Nowadays, there are researches been carried out about apply refactoring in model instead of source code[Ulrich and Newcomb, 2010]. Unfortunately, no catalogue of refactorings for the KDM specification exists. Nevertheless, although ADM provides the process for refactoring legacy systems by means of KDM, there is a lack of an Integrated Development Environment (IDE) to lead engineers to apply refactorings as such exist in others object-oriented languages. Similarly, Model-Driven Refactoring is a special kind of model transformation that allows us to improve the structure of the model while preserving its internal quality characteristics. Model-driven refactoring is a considerably new area of research which still needs to reach the level of maturity attained by source code refactoring [Misbhauddin and Alshayeb, 2012].

In this context, available object-oriented refactoring catalogues are not reusable as they are, because the KDM follow the MDA. This forces developers to create they own refactorings to be applied into models, i.e., they neither follow any catalogue nor use any kind of dedicated support. Given the potential complexity of model-driven refactoring, manual modifications into the models may lead to unwanted side-effects and result in a tedious and error-prone maintenance process. In this sense, the main contribution of this paper is the provision of a refactoring catalogue for KDM. The catalogue is based on our experience as model-driven engineering. Also, the catalogue herein provided is heavily inspired by the refactorings given by Fowler [Fowler et al., 2000].

3. Refactoring for KDM by means of KDM-RE

This sections describes a *plug-in* on the top of the Eclipse Platform named **K**nowledge **D**iscovery **M**odel-**R**efactoring **E**nvironment (KDM-RE). It supports 17 refactorings adapted to KDM. These refactorings are based on some fine-grained refactorings proposed by Fowler [Fowler et al., 2000]. All the adapted refactorings are shown in Table 2. We chose the Fowler's refactorings once them are well known, basic and fine-grained refactorings. Our plugin also uses MoDisco[1] which provides an extensible framework to transform an specific source-code to KDM models.

Table 2. Refactorings Adapted to KDM

N	Refactoring Name	Description
Rename Feature		
1	Rename ClassUnit, StorableUnit, and MethodUnit	A ClassUnit, a StorableUnit or a MethodUnit does not reveal its purpose.
Moving Features Between Objects		
2	Move MethodUnit	A MethodUnit is being using by another ClassUnit than the ClassUnit on which it is defined.
3	Move StorableUnit	A StorableUnit is used by another ClassUnit more than the ClassUnit on which it is defined.
4	Extract ClassUnit	You have one ClassUnit doing work that should be done by two ClassUnit.
5	Inline ClassUnit	A ClassUnit is not doing very much.
Organing Data		
6	Replace data value with Object	You have a data item that needs additional data or behavior.
7	Encapsulate StorableUnit	There is a public StorableUnit.
8	Replace Type Code with ClassUnit	A ClassUnit has a numeric type code that does not affect its behavior.
9	Replace Type Code with SubClass	You have an immutable type code that affects the behavior of a ClassUnit.
10	Replace Type Code with State/Strategy	You have a type code that affects the behavior of a ClassUnit, but you cannot use subclassing.
Dealing with Generalization		
11	Push Down MethodUnit	Behavior on a superclass is relevant only for some of its subclasses.
12	Push Down StorableUnit	A StorableUnit is used only by some subclasses.
13	Pull Up StorableUnit	Two subclasses have the same StorableUnit.
14	Pull Up MethodUnit	You have MethodUnits with identical results on subclasses.
15	Extract SubClass	A ClassUnit has features that are used only in some instances.
16	Extract SuperClass	You have two ClassUnits with similar features.
17	Collapse Hierarchy	A superclass and subclass are not very different.

[1]`http://www.eclipse.org/MoDisco/`

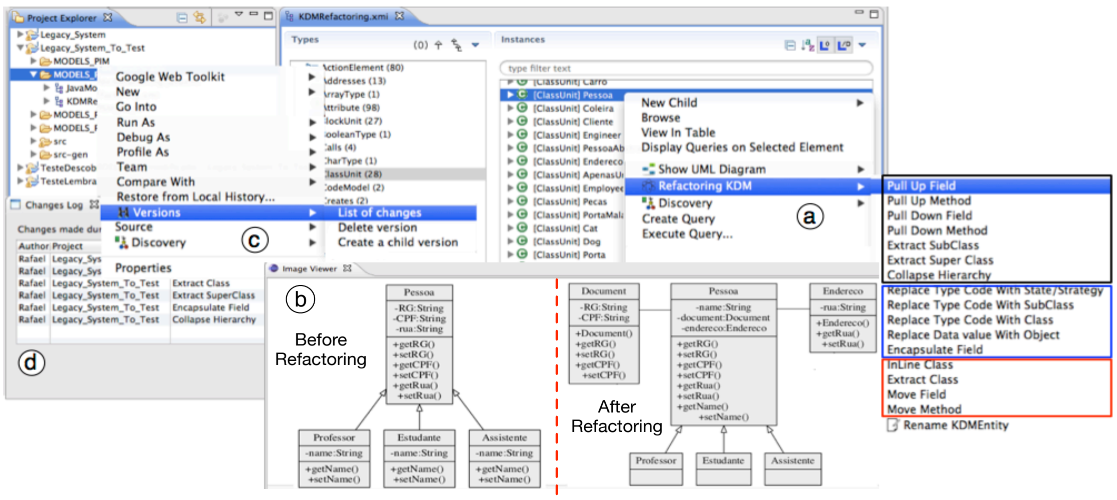


Figure 2. KDM-RE's Interface

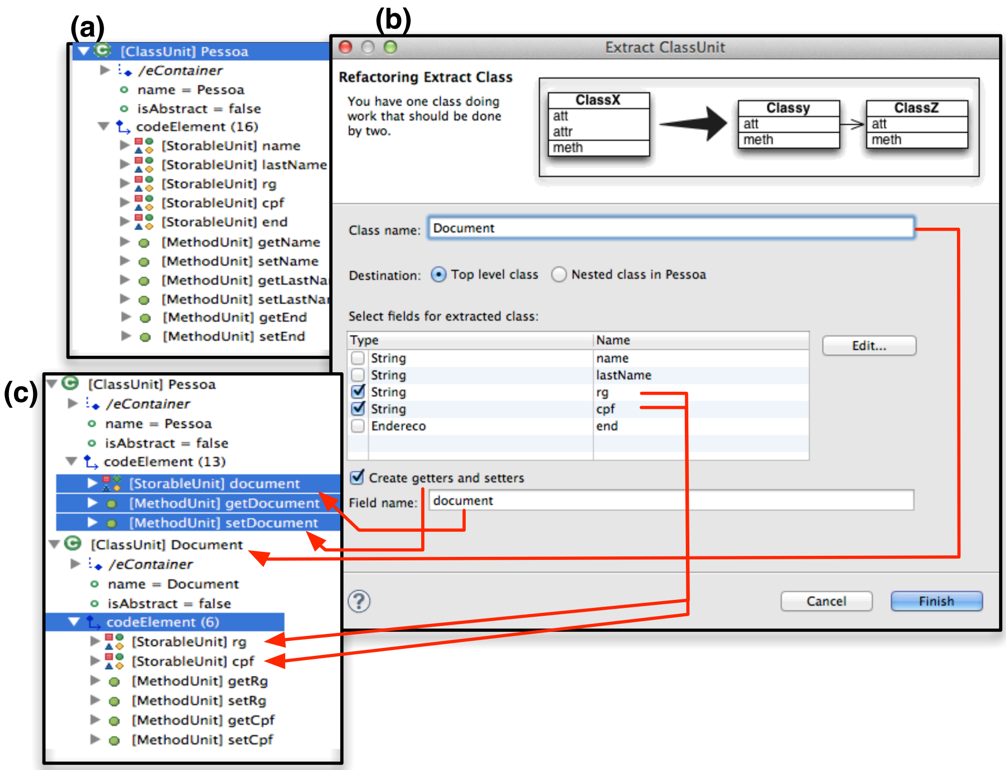
Starting from the popup menu named "Refactoring KDM", in this model browser, see Figure 2ⓐ, either the software developer or software modernizer can interact with the KDM model and choose which refactoring must be carried out in the KDM. In the region ⓐ can be seen all 17 refactorings that have been implemented in KDM-RE. For illustration purposes only we drew rectangles to separate the refactorings into three groups. The black rectangle represents refactorings that deal with generalization, the blue rectangle stand for refactorings to organize data and the red one symbolize refactoring to assist the moving features between objects.

The region ⓑ on Figure 2 shows an UML class diagram that can be used either before to apply some refactorings in order to assist the engineer to decide where to apply the refactorings or this UML class diagram can be generated as the engineer performs the refactorings in KDM model, i.e., changes are reproduced on the fly in a class diagram. We claim that the latter use of this UML diagram is important once the class diagram provides an abstract view of the system, hence, the engineer can visually check the system's changes after applying a set of refactorings. In addition, usually the source code is the only available artifact of the legacy software. Therefore, creating an UML class diagram makes, both the legacy software and the generated software to have a new type of artifact (i.e., UML class models), improving their documentation.

KDM-RE also supplies a multiple versions of a system at level models (KDM) which allows the engineer to work interactively on multiple models and to explore alternate refactoring path. As can see in the region ⓒ (see Figure 2), the engineer must select a KDM file, then he must right-click on the mouse to appear a popup menu named "Versions". By releasing the mouse on this menu, three options is shown: (*i*) List of changes, (*ii*) Delete version and (*iii*) Create a child version. The last option create a copy of the KDM file - then the engineer can explore another refactoring path. The second option delete a specific version - first option shows all changes that have been realized in a current KDM file, all changes are depicted in an Eclipse View, as shown in region ⓓ. In this View it is possible to visualize the author that have committed the changes, the project and all refactorings realized.

3.1. Case Study

In this section, we motivate KDM-RE by analyzing an example. This example is a small part of the university domain. Figure 2 ⓑ (left side) shows a class diagram used for modeling a small part of the university domain. In an university there are several Persons, more specifically Professors, their Assistants, and Students. Each Person has RG, CPF, and address (of type String). Moreover, classes Professor, Assistant, and Student have an attribute name of type String each. Pretend that either the software modernizer or the software developer found out by looking at the UML class diagram (see Figure 2ⓑ left side) this redundantly, i.e., equal attributes in sibling classes. Therefore, he/she must apply the refactoring "Pull Up Field'. Similarly, he/she also found out by looking at the UML class diagram that one class is doing work that should be done by two or more. For example, he/she found that the attributes RG and CPF should be modularized to a class. Similarly, it is necessary to provide more information about they address, such as number, city, country, etc. Therefore, he/she must apply the refactoring "Extract Class" to the attributes "RG", "CPF" and "rua". Due space limitation it is depicted just the extraction of the attributes "RG" and "CPF". The first step is to select the meta-class that he/she identified as a bad smell, i.e., the meta-class to be extracted into a separate one. This step is illustrated in Figure 3(a).


(a)

```
[ClassUnit] Pessoa
  /eContainer
  name = Pessoa
  isAbstract = false
  codeElement (16)
    [StorableUnit] name
    [StorableUnit] lastName
    [StorableUnit] rg
    [StorableUnit] cpf
    [StorableUnit] end
    [MethodUnit] getName
    [MethodUnit] setName
    [MethodUnit] getLastNa
    [MethodUnit] setLastNa
    [MethodUnit] getEnd
    [MethodUnit] setEnd
```

(b)

Extract ClassUnit

Refactoring Extract Class

You have one class doing work that should be done by two.

Class name: Document

Destination: [✓ Top level class] Nested class in Pessoa

Select fields for extracted class:

Type	Name
☐ String	name
☐ String	lastName
☑ String	rg
☑ String	cpf
☐ Endereco	end

Edit...

☑ Create getters and setters

Field name: document

Cancel   Finish

(c)

```
[ClassUnit] Pessoa
  /eContainer
  name = Pessoa
  isAbstract = false
  codeElement (13)
    [StorableUnit] document
    [MethodUnit] getDocument
    [MethodUnit] setDocument
[ClassUnit] Document
  /eContainer
  name = Document
  isAbstract = false
  codeElement (6)
    [StorableUnit] rg
    [StorableUnit] cpf
    [MethodUnit] getRg
    [MethodUnit] setRg
    [MethodUnit] getCpf
    [MethodUnit] setCpf
```

Figure 3. Extract Class Wizard

After selecting the meta-class, a right-click opens the context menu where the refactoring is accessible. After the click, the system displays the "RefactoringWizard" to the engineer, Figure 3(b) depicts the Extract Class Wizard. In this wizard, the name of the new meta-class can be set. Also a preview of all detected `StorableUnits` and

MethodUnits that can be chosen to put into the new meta-class. Further, the engineer can select if either the new meta-class will be a top level meta-class or a nested meta-class. The engineer also can select if the KDM-RE must create instances of MethodUnits to represent accessors methods (gets and sets). Finally, the engineer can set the name of the StorableUnit that represent the link between the two meta-classes (the old meta-class and the new one). After all of the required inputs have been made, the engineer can click on the button “Finish” and the refactoring “Extract Class” is performed by KDM-RE. As can be seen in Figure 3(c) a new instance of ClassUnit named “Document” was created - two StorableUnit from “Pessoa”, i.e., “rg” and “CPF” were moved to the new ClassUnit - instances of MethodUnits were also created to represent the gets and sets. In addition, the instance of ClassUnit named “Pessoa” owns a new instance of StorableUnit that represent the link between both ClassUnits. Due space limitation the other StorableUnits of ClassUnit named “Pessoa” are not shown in Figure 3(c).

After the engineer realize the refactorings, an UML class diagram is created on the fly to mirror graphically all changes performed in the KDM model, see Figure 2ⓑ right side. Moreover, the KDM-RE creates/updates a tracking log to show the historic of all changes performed in the system, as can be see in Figure 2ⓓ.

4. Related Work

Borger et al. [Boger and Sturm, 2002] developed a plug-in for the CASE tool ArgoUML that support UML model-based refactorings. The refactoring of class, states and activities is possible, allowing the user to apply refactorings that are not simple to apply at source code level. Van Gorp et al. [Gorp et al., 2003] proposed a UML profile to express pre and post conditions of source code refactorings using Object Constraint Language (OCL) constraints. The proposed profile allows that a CASE tool: (*i*) verify pre and post conditions for the composition of sequences of refactorings; and (*ii*) use the OCL consulting mechanism to detect bad smells such as crosscutting concerns.

In [Reimann et al., 2010] the authors present an approach for EMF model refactoring. They propose the definition of EMF-based refactoring in a generic way. Another approach for EMF model refactoring is presented in [Thorsten Arendt, 2013], They propose EMF Refactor [2], which is a new Eclipse incubation project in the Eclipse Modeling Project consisting of three main components. Besides a code generation module and a refactoring application module, it comes along with a suite of predefined EMF model refactorings for UML and Ecore models.

The differential of our approach described herein in relation to the other is the proposal to move from software refactoring to model-driven refactoring by means of KDM, which is a platform and language independent meta-model.

5. Concluding Remarks

In this paper is presented the KDM-RE which is a *plug-in* on the top of the Eclipse Platform to provide support to model-driven refactoring based on ADM and uses the KDM standard. More specifically, this *plug-in* supports 17 refactorings adapted to KDM. These

[2]http://www.eclipse.org/emf-refactor/

refactorings are based on some fine-grained refactorings proposed by Fowler [Fowler et al., 2000]. As stated in the case study the engineer/modernizer by using KDM-RE can apply a set refactorings in a KDM. Also, on the fly the engineer can check all changes realized in this KDM replicated into a class diagram - the engineer can visually verify the system‘s changes after applying a set of refactorings. In addition, usually the source code is the only available artifact of the legacy software. Therefore, creating an UML class diagram makes, both the legacy software and the generated software to have a new type of artifact (i.e., UML class models), improving their documentation. KDM-RE also supplies a multiple versions of a system at level models.

We believe that KDM-RE makes a contribution to the challenges of Software Engineering which focuses on mechanisms to support the automation of model-driven refactoring. Future work involves implementing more refactorings and conducting experiments to evaluate all refactorings provided by KDM-RE. Doing so, we hope to address a broader audience with respect to using, maintaining, and evaluating our tools. Notice that KDM-RE is open source and it can be downloaded at *www.dc.ufscar.br/∼valter/KDM-RE*.

6. Acknowledgements

The authors would like to thank FAPESP for Process 2012/05168-4.

References

Boger, M. and Sturm, T. (2002). Tool-support for model-driven software engineering. In *In Proceedings of Practical UML-Based Rigorous Development Methods*.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (2000). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gorp, P. V., Stenten, H., Mens, T., and Demeyer, S. (2003). Towards automating source-consistent uml refactorings. In *International Conference on UML - The Unified Modeling Language*, pages 144–158. Springer.

Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Misbhauddin, M. and Alshayeb, M. (2012). Model-driven refactoring approaches: A comparison criteria. In *Sofware Engineering and Applied Computing (ACSEAC), 2012 African Conference on*.

OMG (2012). Object Management Group (OMG) Architecture-Driven Modernisation. Disponível em: *http://www.omgwiki.org/admtf/doku.php?id=start*. (Acessado 2 de Agosto de 2012).

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois.

Perez-Castillo, R., de Guzman, I. G.-R., Avila-Garcia, O., and Piattini, M. (2009). On the use of adm to contextualize data on legacy source code for software modernization. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE ’09, pages 128–132, Washington, DC, USA. IEEE Computer Society.

Reimann, J., Seifert, M., and Abmann, U. (2010). Role-based generic model refactoring. In *In ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013)*. Springer.

Thorsten Arendt, Timo Kehrer, G. T. (2013). Understanding complex changes and improving the quality of uml and domain-specific models. In *In ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013)*.

Ulrich, W. M. and Newcomb, P. (2010). *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.