Improving the structure of KDM instances via refactorings:

An experimental study using KDM-RE

Rafael S. Durelli
UFLA
rafael.durelli@dcc.ufla.br

Matheus C. Viana
UFSJ
matheuscviana@ufsj.edu.br

André de S. Landi
UFSCar
andre.landi@dc.ufscar.br

Vinicius H. S. Durelli
UFSJ
durelli@ufsj.edu.br

Marcio E. Delamaro
USP
delamaro@icmc.usp.br

Valter V. de Camargo
UFSCar
valter@dc.ufscar.br

ABSTRACT

Architecture-Driven Modernization (ADM) is an initiative of the Object Management Group (OMG) whose main purpose is to provide standard metamodels for software modernization activities. The most important metamodel is the Knowledge Discovery Metamodel (KDM), which represents software artifacts in a language-agnostic fashion. A fundamental step in software modernization is refactoring. However, there is a lack of tools that address how refactoring can be applied in conjunction with ADM. We developed a tool, called KDM-RE, that supports refactorings in KDM instances through: (*i*) a set of wizards that aid the software modernization engineer during refactoring activities; (*ii*) a change propagation module that keeps the internal metamodels synchronized; and (*iii*) the selection and application of refactorings available in its repository. This paper evaluates the application of refactorings to KDM instances in an experiment involving seven systems implemented in Java. We compared the pre-refactoring versions of these systems with the refactored ones using the Quality Model for Object-Oriented Design (QMOOD) metric set. The results from this evaluation suggest that KDM-RE provides advantages to software modernization engineers refactoring systems represented as KDMs.

CCS CONCEPTS

• **Software and its engineering** → *Software notations and tools*;

KEYWORDS

Architecture-Driven Modernization, Knowledge-Discovery Metamodel, Refactoring, Model-Driven Development

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SBES'17, September 20–22, 2017, Fortaleza, CE, Brazil
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5326-7/17/09...$15.00
https://doi.org/10.1145/3131151.3131153

ACM Reference format:
Rafael S. Durelli, Matheus C. Viana, André de S. Landi, Vinicius H. S. Durelli, Marcio E. Delamaro, and Valter V. de Camargo. 2017. Improving the structure of KDM instances via refactorings:. In *Proceedings of SBES'17, Fortaleza, CE, Brazil, September 20–22, 2017,* 10 pages.
https://doi.org/10.1145/3131151.3131153

1 INTRODUCTION

Legacy systems are software systems that are still useful to support the internal processes of an organization, despite posing maintenance and evolution challenges. The structure of a legacy system is usually inconsistent with its documentation, which makes it difficult to maintain. However, replacing it completely is often expensive and error-prone task [24].

An alternative to replacing a legacy system is reengineering. Reengineering entails analyzing the system source-code (or its user interface) to create higher levels models of the software system under investigation and then redesign and rebuild those models [24]. In 2003, the Object Management Group (OMG) released the concept of Architecture-Driven Modernization (ADM), whose main objective is to standardize reengineering processes through metamodels [21]. The Knowledge Discovery Metamodel (KDM) is the primary metamodel of an ADM process. It has a vast amount of metaclasses to represent lower levels of abstraction of a system (e.g., source-code), higher levels (e.g., architecture, business rules, and other abstract concepts), and technical levels (e.g., graphic interface, configuration files, and databases). KDM allows the representation of concepts of any domain [9].

Refactoring is a key activity in an ADM process. It was first proposed as a methodology to restructure programs by Opdyke [22]. After that, refactoring became a widely adopted discipline to improve software systems without changing its observable behavior [14]. With appropriate tooling support, refactoring can be an efficient and effective way to (*i*) improve software design and (*ii*) make software easier to understand.

Although refactoring is a well-known concept when applied to source-code, there are some research issues when this concept is applied to models [30]. Model-level refactoring tends to be more complex than source-code refactoring because, apart from restructuring model elements and relationships, it is also necessary to check whether the model remains consistent with the source-code and keep track of the

synchronization between them [17, 18]. On the other hand, one of the advantages of using model-based refactoring is the fact that software developers do not have to worry about programming-language specific features, abstracting away low-level, programming-language constructs. Models provide a graphical and high-level view of the systems, thus software engineers can easily visualize and verify which refactorings should be applied to the system.

When it comes to model-based refactoring, most research efforts present proposals for applying refactorings to Unified Modeling Language (UML) diagrams [4, 13, 19, 27, 28, 30], since UML is widely used to design and document software system. However, using only UML as a base metamodel in modernization tools restricts the restructuring activities to the views available in this metamodel. There are several system modernization scenarios that require visions and representations that extrapolate UML diagrams, but can be properly represented with KDM [4, 13, 19, 27, 28, 30]. For instance, UML does not contain a set of dedicated, specific metaclasses to represent lower levels of abstraction, such as source-code, neither metaclasses to represent higher levels, such as domain concepts, architecture, business rules, and user interfaces.

The complexity of restructuring activities renders them unwieldy. There are tools that allow the application of refactorings to class diagrams. However, none of them uses KDM as their underlying metamodel. As opposed to UML models, KDM instances can store many details about a system but these metamodels were not designed to serve as a graphical view of the system. Thus, in order to make it possible to apply refactorings to KDM instances, UML can be used as a graphical representation of KDM.

Due to the lack of tools working with KDM, in a previous paper [8], we presented a tool, called Knowledge Discovery Model-Refactoring Environment (KDM-RE), which aids software engineers to refactor UML class diagrams and automatically propagates these changes to KDM instances. This tool, which is a plug-in for Eclipse IDE, automates the entire process of application, reuse, and propagation of refactorings to KDM instances, so that software engineers only need to identify where to apply those transformations.

As a follow-up to previous research, in this paper we set out to evaluate whether the refactorings created for KDM improve system quality. To this end, we carried out an experiment in which we used KDM-RE to apply the refactorings to seven open-source systems implemented in Java. After applying all the transformations, we compared the pre-refactoring versions of these systems with the refactored versions using the metrics proposed by QMOOD. The results of our evaluation suggest that refactoring transformations applied to KDM instances improve some quality-related attributes such as reusability, flexibility, understandability, and effectiveness. Moreover, KDM-RE provides advantages to software modernization engineers refactoring systems represented as KDMs.

The remainder of this paper is organized as follows. Section 2 provides background on KDM. Section 3 presents KDM-RE. Section 4 presents the experiment that we conducted to evaluate the benefits of the refactoring transformations that act upon KDM instances. Section 5 describes our experimental results and presents related work. Section 6 provides concluding remarks.

2 ADM AND KDM

ADM [23] is an OMG initiative for standardizing system modernization processes. The main idea is taken advantage of reverse engineering concepts, Model-Driven Architecture (MDA) principles and the KDM meta-model. A typical ADM modernization process starts by reverse engineering a system into a KDM instance, this instance is then analyzed in hopes of finding problems, afterwards, refactorings are applied to the resulting KDM instance. The process ends with the generation of the modernized system. According to Perez [23], there are several modernization scenarios that can be conducted to modernize legacy systems: platform migration, language to language conversion, and application improvement. The last one involves architecture reconstruction, which is the scenario we approach in this paper.

KDM is a language and platform independent ISO metamodel capable of representing a complete software system. KDM can be seen as a family of meta-models and it comprises several metamodels for representing systems inside modernization tools. A schematic representation of KDM can be seen in Figure 1. It is divided into four layers (right side) that are further divided into packages (internal meta-models). Each package concentrates on specific aspects of software systems. Thus, there are packages (meta-models) for representing a wide spectrum of system abstractions, from low-level details of source-code (`Code` package) and run-time actions (`Action` package), to user interface (`UI` package), deployment details (`Build` package), Business Rules abstractions (`Conceptual` package), and Architectural details (`Structure` package).

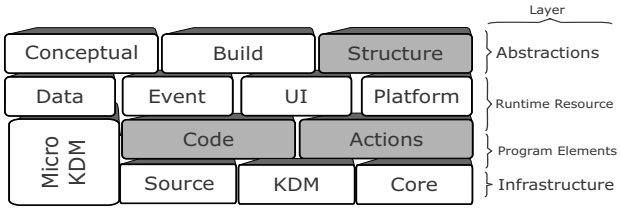

Figure 1: KDM layers.

`Code`, `Action` and `Structure` are the most important packages in the context of our research because they represent all artifacts upon which our tool performs refactorings. Figure 2 shows a small snippet of KDM, showing some metaclasses of the `Structure`, `Code`, `Core` and `Action` packages.

`Core` is central for KDM, since it provides base metaclasses for other packages. `KDMEntity` is the most important metaclasses in this package because all other KDM metaclasses

directly or indirectly extend it. Therefore, all KDM metaclasses are KDMEntities. For instance, `Code` contains a lot of metaclasses for representing code details, such as `ClassUnit`, `MethodUnit`, and `Package`. `Code` contains 24 metaclasses to represent all statements in a object oriented programming language. Table 1 shows some of these metaclasses.

Table 1: `Code` and its metaclasses.

Statements	`Code` metaclasses
Class	`ClassUnit`
Interface	`InterfaceUnit`
Method	`MethodUnit`
Attribute	`StorableUnit`
Local Variable	`MemberUnit`
Parameter	`ParameterUnit`
Association	`KDMRelationShip`

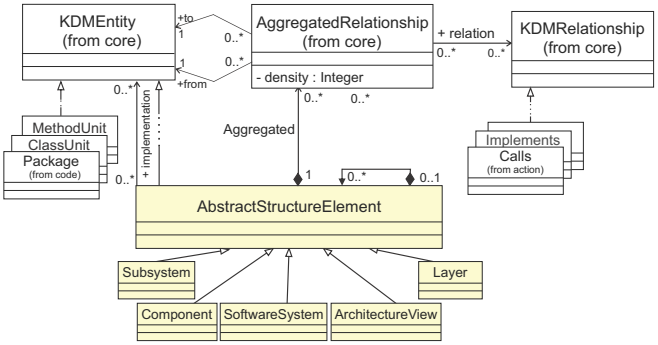

Figure 2: Class diagram of the `Structure` package (OMG Group [23]).

`AggregatedRelationship` is another important metaclass. It is a relationship that allows to group other primitive relationships within it. This is represented by the association with the `KDMRelationship` class. In KDM, every relationship type is represented by a metaclass. Instances of primitive relationships are method calls (`Calls` metaclass), object instantiations (`Creates` metaclass), implements relationships (`Implements` metaclass), etc. Each `AggregatedRelationship` involves two KDM Entities: source and target.

3 KDM-RE

Figure 3 shows KDM-RE logical architecture, which is divided into three layers: (*i*) Integrated Development Environment (IDE), (*ii*) KDM-RE, and (*iii*) User Interface (UI). The first layer contains all Eclipse IDE plug-ins that KDM-RE depends on: Eclipse Modeling Framework (EMF) [10], ATL, Object Constraint Language (OCL), Modisco [11], Papyrus[1], XText[2], EMFCompare[3], and KDM.

The second layer contains the three modules of KDM-RE: (*i*) Refactoring Module, (*ii*) Structured Refactoring Meta-Model (SRM) Module, and (*iii*) Synchronization Module. In this paper, we focus in the first module, which contains a set of resources that automate the application of refactoring transformations to KDM instances.

The third layer encompasses KDM-RE perspective on Eclipse IDE graphical interface. It provides two graphical editors: the first one is an extension of the editor MoDisco [5] and where KDM instances are shown, and the second graphical editor is an extension of Papyrus [12] that shows UML class diagrams.

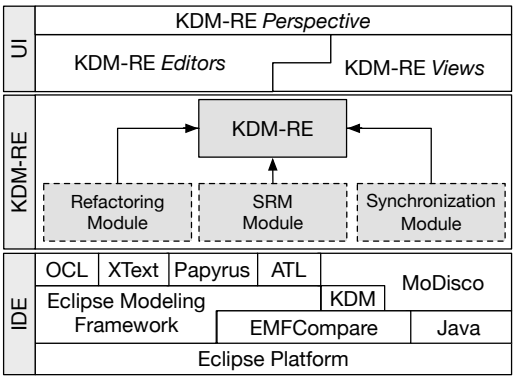

Figure 3: Architecture of KDM-RE.

Refactorings can be grouped according to their level of granularity [14]. Granularity can be defined in two levels of operations: *(i)* atomic operations, and *(ii)* compound operations. Atomic operations are specified by the following primitive operations:

- `ADD`: any operation that adds an instance of a KDM metaclass, for instance: `ADD Package`, `ADD ClassUnit`, `ADD StorableUnit`, `ADD MethodUnit`, etc.
- `DELETE`: any operation that removes an instance of a KDM metaclass, for instance: `DELETE Package`, `DELETE ClassUnit`, `DELETE StorableUnit`, `DELETE MethodUnit`, etc.
- `CHANGE`: any operation that changes a value of a meta-attribute of a KDM metaclass, for instance: `CHANGE Package meta-attributes`, `CHANGE ClassUnit meta-attributes`, `CHANGE StorableUnit meta-attributes`, `CHANGE MethodUnit meta-attributes`, etc.

Compound operations consist of a combination of atomic operations. According to Fowler [14], the major refactorings are defined using atomic operations (`ADD`, `DELETE`, and `CHANGE`). Therefore, by combining these atomic operations it is possible to create more complex refactorings. For instance, Table 2 describes the operations that are combined to create more sophisticated refactoring transformations.

All refactorings were defined in KDM-RE using ATL. Internally KDM-RE contains templates for each atomic operations (`ADD`, `DELETE`, and `CHANGE`) devised in ATL. Listing 1 presents the templates for the atomic operations `ADD` (lines 1-11), `DELETE` (lines 13-19), and `CHANGE` (21-28), respectively. The fixed parts of the templates are formed by ATL and the

[1]https://eclipse.org/papyrus/
[2]https://eclipse.org/Xtext/
[3]https://www.eclipse.org/emf/compare/

varying parts, e.g., the dashed line, are parameters to execute the refactorings. KDM-RE programmatically executes the transformations implemented in ATL by means of ATL EMF Transformation Virtual Machine.[4] Similarly, all pre- and postconditions were specified using OCL. The API Desden OCL[5] was used to execute all pre- and postconditions.

Listing 1: Template ATL to perform the atomic operation ADD

```
rule ADD {
 from
  source : MM!____ (source.name = ____)
 to
  target: MM!____ (
   codeElement ← source.codeElement→including(new)
  ),
  new: MM!____ (
   name ← ____
  )
}

rule DELETE {
 from
  source : MM!____ (source.name = ____ and
        source.refImmediateComposite().name = ____)
 to
  drop
}

rule CHANGE {
 from
  source : MM!____ (source.name=____)
 to
  target : MM!____ (
   ____ ← ____
 )
}
```

All refactoring transformations supported by KDM-RE are applied to KDM instances. As a result, KDM-RE integrated the core functionality of MoDisco, which is an Eclipse plug-in capable of transforming Java source code into KDM instances. Listing 2 shows a chunk of code of how KDM-RE uses the MoDisco to transform a Java project into a KDM instance.

Listing 2: Transforming Java source-code into KDM instance

```
DiscoverSourceModelFromJavaElement discoverer = ...;
discoverer.discoverElement(javaProject, monitor);
Resource kdmModel = discoverer.getTargetModel();
```

After obtaining a KDM instance, software engineers can apply the refactorings using KDM-RE graphical editors (GE). In KDM-RE, KDM instances can be represented in a tree model browser, as shown in Figure 4. The left side shows all KDM metaclasses instantiated in a Java project. The right side describes all meta-attributes of an specific KDM's metaclass – one can right click in any of these metaclasses in order to apply the refactoring (Figure 4) [8]. Although this GE is useful for applying refactoring directly to KDM metamodel, it is not intuitive.

[4]https://wiki.eclipse.org/ATL/EMFTVM
[5]http://www.dresden-ocl.org/index.php/DresdenOCL

Table 2: Refactoring transformations and their constituent atomic operators.

Refactoring	Operations
Move StorableUnit	ADD | DELETE
Move MethodUnit	ADD | DELETE
Extract ClassUnit	ADD | DELETE | CHANGE
Inline ClassUnit	CHANGE | DELETE
Flatten Hierarchy	ADD | DELETE | CHANGE
Push Down MethodUnit	ADD | DELETE
Push Down StorableUnit	ADD | DELETE
Pull Up MethodUnit	ADD | DELETE
Pull Up StorableUnit	ADD | DELETE
Extract SubClassUnit	ADD | DELETE
Encapsulate StorableUnit	ADD | CHANGE

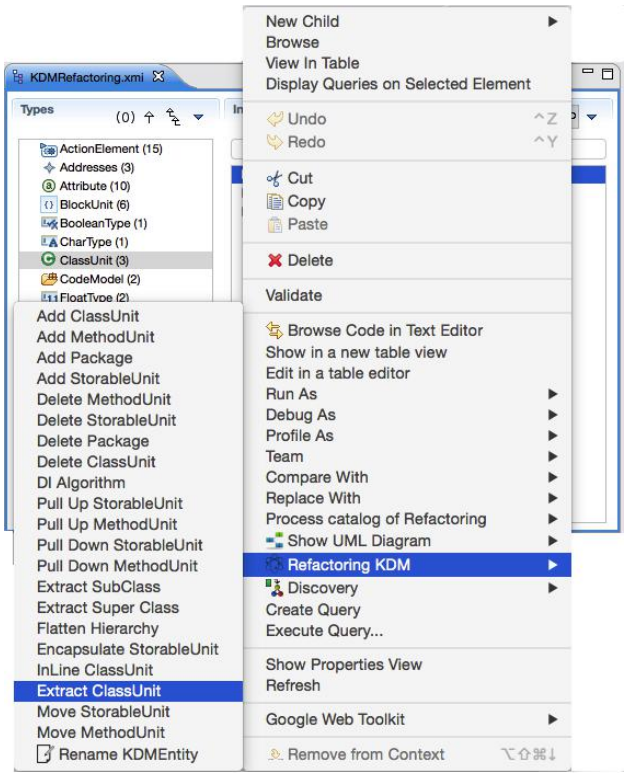

Figure 4: KDM-RE Tree Model Browser.

KDM-RE also allows (Figure 5 (A)), the software engineer to apply refactorings by using UML class diagram. Although this editor uses UML class diagram, all refactorings are in fact executed in KDM instances, UML class diagrams are used as bridges between the information (e.g., class name, attributes, methods) and the refactorings. After choosing the refactoring in both GUIs an specific `RefactoringWizard` is launched, see Figure 5 (B). It guides the software engineer throughout the refactoring process. An important feature of KDM-RE is the option to preview the result of applying a given refactoring. Thus, if the software engineer wants to visualize the effect of refactoring before actually performing it, he can select the `Preview` button. After clicking on `Preview`, a screen contrasting the before and after version of the system will be created as shown in Figure 6. As shown in Figure 6, the top of the screen shows which instances were deleted, moved, and added in a textual way. In the bottom, it is possible to visualize the difference between the two KDM instances, that is, the non-refactored (original) instance and the refactored instance. The right side represents the KDM

instance after the application of a refactoring and the left side represents the KDM instance before refactoring. We have used EMFCompare[6] to implement this feature.

4 EVALUATION

We carried out an experiment to evaluate whether the refactorings applied to KDM instances contribute to improve the quality of systems. Table 3 presents the seven systems used in the experiment: Xerces-J, Jexel, JFreeChart, JUnit, GanttProject, ArtofIllusion, and JHotDraw. These seven systems were chosen because they are real-world Java applications whose sizes range from 16,026 to 240,540 lines of code. Xerces-J is a software family for parsing XML; Jexel is API for writing regular expressions in Java; JFreeChart is a Java library used to generate charts; JUnit is a framework used to generate unit tests; GanttProject is a system for project management; ArtofIllusion is an API for 3D modeling and rendering; JHotDraw is a tool to aid the creation of drawings.

As mentioned, in this experiment we set out to verify whether, after applying a set of refactorings to remove some bad smells, the systems improved in terms of quality. More specifically, we evaluated the quality of the subject systems according to a set of metrics: QMOOD, which is a quality model for object-oriented programs that establishes an empirically validated hierarchical structure to evaluate quality attributes [2]. We decided to use QMOOD because (*i*) this set of metrics is widely used in the literature to evaluate the impact of refactorings on software [15, 20]; and (*ii*) QMOOD defines six quality attributes that can be gauged by the metrics [2]. In this experiment, we examined four QMOOD quality attributes: Reusability (R), Flexibility (F), Understandability (U), and Effectiveness (E). The quality attribute functionality was not taken into account in our experiment because refactorings by definition should not change the observable behavior of software. Extensibility was not taken into consideration due to the subjectivity associated with this quality attribute.

Table 4 presents the system properties that were analyzed in the KDM instances and their respective metrics, as defined in QMOOD [2]. The relationship between theses metrics and the quality attributes is shown in Table 5. In our experiment, a system has a quality gain in a given attribute (G_q) when: $G_{qi} = q'_i - q_i$, where q_i and q'_i are the value of quality attribute i before and after the behavior preserving transformations, respectively.

Table 3: Systems used in the experiment.

System	Abbreviation	Version	KLOC	Classes
Xerxes-J	XJ	2.7.0	240	991
Jexel	JEX	1.3	50.4	75
JFreeChart	JFC	4.0	170	521
JUnit	JUn	4.0	17.48	225
GanttProject	GP	1.10.2	41	245
ArtofIllusion	AOIL	2.8.1	87	459
JHotDraw	JHD	7.0.6	16	468

[6]https://www.eclipse.org/emf/compare/

Table 4: Metrics used in the experiment.

Property	Metric
Design size	Design Size in Classes (DSC)
Hierarchies	Number Of Hierarchies (NOH)
Abstraction	Avarage Number of Ancestors (ANA)
Encapsulation	Data Access Metric (DAM)
Coupling	Direct Class Coupling (DCC)
Cohesion	Cohesion Among Methods in class (CAM)
Composition	Measure Of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number Of Polymorphic methods (NOP)
Complexity	Number Of Methods (NOM)
Messaging	Class Interface Size (CIS)

Table 5: Relationship between quality attributes and the metrics [2].

R	-0.25DCC + 0.25CAM + 0.5CIS + 0.5DSC
F	0.25DAM - 0.25DCC + 0.5MOA + 0.5NOP
U	-0.33ANA + 0.33DAM -0.33DCC + 0.33CAM - 0.33NOP + 0.33NOM - 0.33DSC
E	0.2ANA + 0.2DAM + 0.2MOA + 0.2MFA + 0.2NOP

The experiment was designed following the approach defined by Wohlin et al. in [31], which consists of three steps. The first step, definition and planning, encompasses context specification, hypotheses formulation, operational definition of variables, and description of participants (when applicable). The second step, termed operation, involves defining details related to the experiment preparation and execution. The third step, which is data analysis, has to do with defining how the information collected during the previous step will be analyzed. Subsections 4.1 to 4.3 describe each step of the experiment.

4.1 Definition and Planning

We defined the experiment following the Goal, Question, Metric (GQM) model [31]:

- **Object of study**: the refactorings applied to KDM instances;
- **Purpose**: to evaluate the impact of refactorings performed by KDM-RE on KDM instances;
- **Focus**: reusability, flexibility, understandability, and effectiveness, as defined in QMOOD [2];
- **Perspective**: software engineers;
- **Context**: this experiment was conducted using the Eclipse IDE 4.3.2, on a 2.5 GHz Intel Core i5 machine with 8GB of physical memory running the Mac OS X 10.9.2 operating system.

The experiment can be summarized using the following template [31]: **to analyze** the refactorings created for KDM; **with the purpose of** evaluating the proposed refactorings; **with respect to** reusability, flexibility, understandability, and effectiveness; **from the point of view of** software engineers; **in the context** of different systems. In order to achieve this, the following research question was defined:

Research Question (RQ): How can refactorings created for KDM instances be useful to software engineers in real world scenarios?

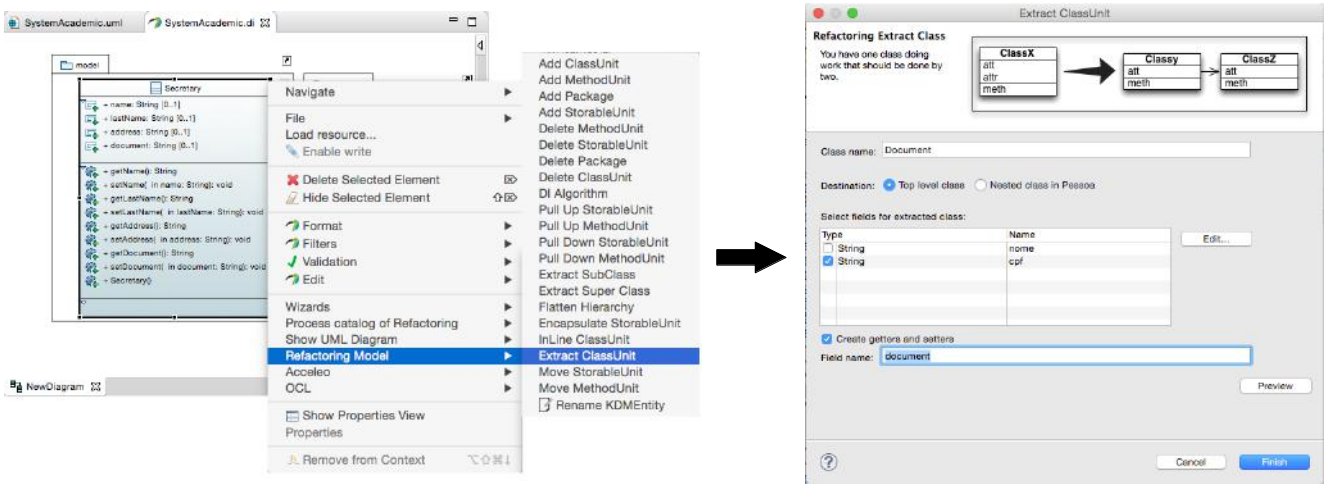

Figure 5: KDM-RE UML Class Diagram.

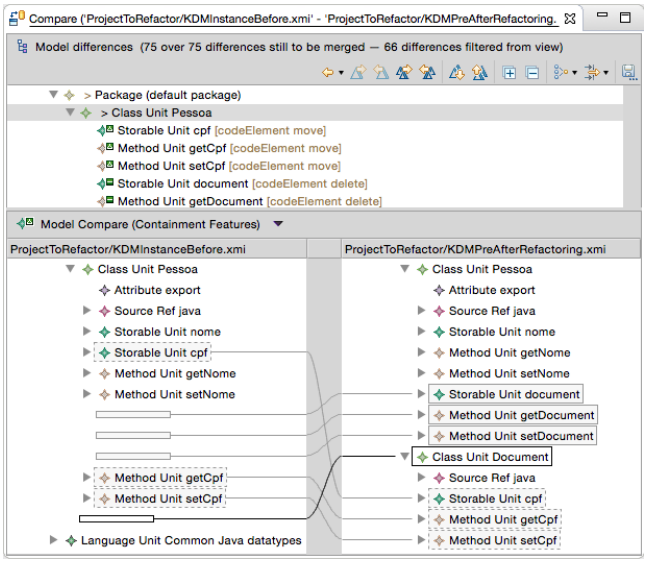

Figure 6: Refactoring Preview.

The RQ was formalized into the following hypotheses created for each quality attribute:

- **Reusability**:
  - **Null Hypothesis** ($H1\_0$): regarding reusability, there is no difference between KDM instances before and after refactoring:
    $H1_0 : \mu_{reusability_{before}} = \mu_{reusability_{after}}$
  - **Alternative Hypothesis** ($H1\_1$): int terms of reusability, there is a difference between KDM instances before and after refactoring:
    $H1_1 : \mu_{reusability_{before}} \neq \mu_{reusability_{after}}$
- **Flexibility**:
  - **Null Hypothesis** ($H1\_0$): regarding flexibility, there is no difference between KDM instances before and after refactoring:
    $H2_0 : \alpha_{flexibility_{before}} = \alpha_{flexibility_{after}}$
  - **Alternative Hypothesis** ($H1\_1$): regarding flexibility, there is a difference between KDM instances before and after refactoring:
    $H2_1 : \alpha_{flexibility_{before}} \neq \alpha_{flexibility_{after}}$
- **Understandability**:
  - **Null Hypothesis** ($H1\_0$): regarding understandability, there is no difference between KDM instances before and after refactoring:
    $H3_0 : \beta_{understandability_{before}} = \beta_{understandability_{after}}$
  - **Alternative Hypothesis** ($H1\_1$): concerning understandability, there is a difference between KDM instances before and after refactoring:
    $H3_1 : \beta_{understandability_{before}} \neq \beta_{understandability_{after}}$
- **Effectiveness**:
  - **Null Hypothesis** ($H1\_0$): regarding effectiveness, there is no difference between KDM instances before and after refactoring:
    $H4_0 : \gamma_{effectiveness_{before}} = \gamma_{effectiveness_{after}}$
  - **Alternative Hypothesis** ($H1\_1$): as for effectiveness, there is a difference between KDM instances before and after refactoring:
    $H4_1 : \gamma_{effectiveness_{before}} \neq \gamma_{effectiveness_{after}}$

The experiment had the following independent variables: (*i*) the KDM-RE tool; (*ii*) the tools JDeodorant, inFusion, and EMF Metrics; (*iii*) the Java programming language; and (*iv*) the seven systems we chose (Table 3). The dependent variables are (*i*) reusability, (*ii*) flexibility, (*iii*) understandability, and (*iv*) effectiveness.

4.2 Operation

In this section was divided into two parts: preparation and execution.

4.2.1 Preparation. KDM-RE only helps the software engineer to perform the refactorings, it does not identify which refactorings are necessary. Therefore, we used the tools Jdeodorant[7] and inFusion[8] to identify bad-smells [14] in the source code of the systems shown in Table 3. Then, we obtained a list with the following bad-smells: *God Class*, a class with many responsibilities; *Data Class*, a class that only has attributes, but do not use them anywhere; *Spaghetti Code*, a tangled source-code with complex control structures; and Functional Decomposition, when a class plays a single function rather than encapsulating data and functionality.

The next step was to obtain a KDM instance of each system using MoDisco. We also used the tool EMF Metrics [1] on the KDM instances to obtain the measures of the metrics shown in Table 4. Afterwards, we calculated the value of the quality attributes according to the equations in Table 5.

4.2.2 Execution. After identifying the bad-smells of each system, we applied the proper refactorings on the KDM instances of the systems using the KDM-RE tool. The refactorings chosen to remove bad-smells and improve quality attributes are not KDM-specific. They were carefully identified in the refactoring catalog proposed by Fowler [14].

Since applying the refactorings manually would take a long time and demand a lot of effort, a script file with the refactorings, parameters, and paths to KDM the instances was created. Thus, the KDM-RE tool performed the refactorings in a semi-automatic way. For each refactoring, a time limit of 5 minutes was set. It was necessary to prevent the KDM-RE from getting stuck into an infinite loop. The list of refactorings applied to each system is shown in Table 6.

After applying the refactorings to the seven systems, the EMF Metrics tool was used again to measure the quality attributes. The analysis of data before and after refactorings is presented in the next section.

4.3 Data Analysis

Table 7 presents the results obtained from the application of the formula shown in Table 5 with quality attributes calculated before and after the refactorings presented in Table 6. The same data is plotted in the bar graphs shown in Figure 7.

As shown in Table 7, all quality attributes were improved after the refactorings. It is worth nothing that understandability was the quality attribute that improved the most, approximately 18%. On the other hand, effectiveness had the lowest gain, reaching 5.29%. This is mainly because most of the refactorings (*Move Method*, *Move Field*, and *Extract Class*) increased the metrics of coupling (DCC), cohesion (CAM), and project size (DSC), which are used to calculate the attribute of quality understandability. In addition, the

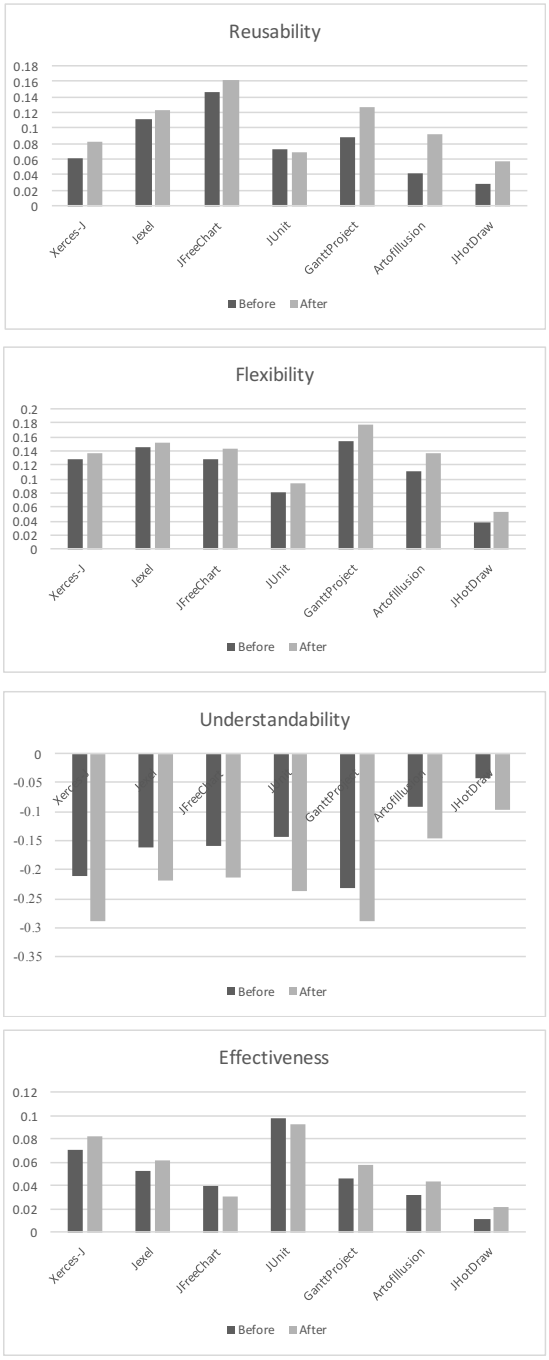

Figure 7: Graphs of the experiment data.

JHotDraw system produced the smallest increase for the four quality attributes. The reason for this is that JHotDraw was originally developed precisely to exemplify the use of good design practices [16], mainly design patterns, so few refactorings needed to be applied to this system.

[7] https://marketplace.eclipse.org/content/jdeodorant
[8] https://www.intooitus.com/products/infusion

Table 6: Type and amount of refactorings applied to each system.

	System								
Refactoring	XJ	JEX	JFC	JUn	GP	AOIL	JHD	Average	Percentage
Move Method	40	30	45	35	43	45	23	37.29	26.34%
Move Field	25	15	20	16	21	23	13	19.00	13.42%
Extra Class	15	20	20	19	12	15	11	16.00	11.30%
Extract Interface	13	10	15	16	8	12	10	12.00	8.48%
Move Class	9	11	7	13	10	10	18	11.14	7.87%
Pull Up Field	12	14	5	8	12	11	9	10.14	7.16%
Pull Up Method	10	9	8	6	8	14	13	9.71	6.86%
Push Down Method	17	13	15	10	9	16	8	12.57	8.88%
Push Down Method	16	12	14	11	10	18	15	13.71	9.69%
TOTAL	157	134	149	134	133	164	120	141.57	100.00%

Table 7: Experiment data before and after the refactorings.

	Reusability		
System	Before	After	Difference
XJ	0.061	0.082	0.021
JEX	0.112	0.124	0.012
JFC	0.147	1.161	0.014
JUn	0.072	0.068	-0.004
GP	0.089	0.127	0.038
AOIL	0.041	0.092	0.051
JHD	0.028	0.057	0.029
Average	0.078	0.094	0.023
Percentage	43.62%	56.38%	12.77%
	Flexibility		
System	Before	After	Difference
XJ	0.128	0.136	0.008
JEX	0.145	0.152	0.007
JFC	0.129	0.143	0.014
JUn	0.081	0.094	0.013
GP	0.153	0.178	0.025
AOIL	0.111	0.136	0.025
JHD	0.039	0.054	0.015
Average	0.112	0.172	0.015
Percentage	46.81%	53.19%	6.37%
	Understandability		
System	Before	After	Difference
XJ	-0.21	-0.289	-0.079
JEX	-0.162	-0.218	-0.056
JFC	-0.159	-0.213	-0.054
JUn	-0.143	-0.236	-0.093
GP	-0.231	-0.289	-0.058
AOIL	-0.093	-0.147	-0.054
JHD	-0.042	-0.098	-0.054
Average	-0.148	-0.212	-0.064
Percentage	41.11%	58.89%	17.79%
	Effectiveness		
System	Before	After	Difference
XJ	0.071	0.082	0.011
JEX0.052	0.052	0.061	0.009
JFC	0.04	0.031	-0.009
JUn	0.097	0.092	-0.005
GP	0.046	0.057	0.011
AOIL	0.032	0.043	0.011
JHD	0.011	0.022	0.011
Average	0.049	0.055	0.005
Percentage	47.35%	52.65%	5.29%

Table 8: Results of the Shapiro-Wilk test.

Before		After	
Reusability			
w=0.97	p-value=0.8988	w=0.9494	p-value=0.7245
Flexibility			
w=0.8998	p-value=0.3298	w=0.9129	p-value=0.416
Understandability			
w=0.9594	p-value=0.8137	w=0.9203	p-value=0.4716
Effectiveness			
w=0.9756	p-value=0.9355	w=0.9621	p-value=0.8362

4.3.1 Test of the Hypotheses. After data collection, we applied statistical tests to the experiment results. Firstly, we checked whether or not the data sets followed a normal distribution applying the Shapiro-Wilk test to each data set (the quality attributes before and after the refactorings, as shown in Table 7). Table 8 presents the resulting *p-value* for each quality attribute. As every *p-value* was greater than 0.05, it can be stated, with a confidence level of 95%, that all data sets follow a normal distribution.

Given that our data follows a normal distribution, we applied the paried t-test to verify our hypotheses. As shown in Table 9, three quality attributes (reusability, flexibility, and understandability) had a $p\text{-}value < 0.05$. Therefore, with 95% confidence level, for these three attributes, there is evidence that the proposed refactorings have a positive impact on these quality attributes.

Table 9 also shows that, statistically, the quality attribute of effectiveness did not improved much. In other words, we could not refute the null hypothesis ($H4_0$) for this quality attribute. We surmise that this happened because understandability and effectiveness affect each other in a way that, when one is improved, the other tends to get worse [2]. As it can be seen in Table 7, after the refactorings, understandability was the quality attribute with the highest increase (17.79%), while effectiveness was the one with the lowest increase (5.29%). This sort of trade-off has to be analyzed by the developers, who have to keep in mind the priorities of their systems when a quality attribute improve to the detriment of another one.

4.3.2 Threats to Validity. **External validity**: refers to the generality of the experiment. The experiment was conducted with seven different and widely used open source systems, belonging to different domains and with different sizes. However, it is not possible to state that the results can be generalized for all Java applications instantiated using KDM, as well as for other programming languages, and other software engineers. Another threat may be the limited number of systems, which externally threatens the generalization of the results of the present study. Future replication of this study is necessary to confirm the outcome of the approach.

Validity by Construction: is concerned with the relationship between theory and what is observed. Since the KDM-RE tool in its current state does not identify bad-smells, the identification of the refactorings to be applied was totally dependent on the Jdeodorant and inFusion tools. Therefore, it is no possible to eliminate validity threats related to problems while identifying bad-smells.

Table 9: Results of the Paried T-Test.

Quality Attribute	T	P-Value	Result
Reusability	-3.3498	0.01542	H1_{0} is refuted due less significance than 5%
Flexibility	-5.5602	0.001433	H2_{0}is refuted due less significance than 5%
Understandability	11.0194	3.322e-05	H3_{0} is refuted due less significance than 5%
Effectiveness	-1.6951	0.141	H4_{0} is not refuted because 0.141 >0.05

Conclusion Validity: is related to the accuracy of the metrics used in the experiment. To mitigate this threat, we used metrics consolidated in the literature [2]. Another possible threat to the validity of our study is concerned with the granularity of the refactorings automated by KRM-RE (i.e., they are not concerned with improving architectural elements/-constructs). However, it is worth emphasizing that, theoretically, it is possible to improve any given software system via small refactorings (i.e., as the ones originally proposed by Fowler [14]). Therefore, small refactorings can be seen as "building blocks" for larger restructuring of software systems, i.e., large refactorings. Consequently, applying two or more low-granularity refactorings can indeed yield architecture-level changes. In addition, although our tool implements only low-level refactorings, the modifications brought about by these refactorings are propagated to other abstraction levels of the metamodel.

5 RELATED WORK

Arendt and Taentzer [1] created a tool called EMF Refactor to apply refactorings in EMF models. EMF Refactor allows to perform identification of bad smells and then the user can apply refactorings. Similarly to KDM-RE, a set of refactorings proposed by Fowler [14] was implemented in EMF Refactor. There are two main differences between KDM-RE and EMF Refactor. The first is that EMF Refactor does not worry about synchronizing the instances of a metamodel after the refactorings; Unlike KDM-RE, which defines a unique synchronization module for this feature. The second difference is related to the identification of which refactoring to apply.

Another tool related to KDM-RE is Refactory, which can refactor any metamodel defined by the EMF. According to Reimann et al. [25], Refactory was created to allow generic refactorings. The main similarities with KDM-RE are: (i) Refactory also allows to apply refactorings graphically through UML diagrams, (ii) refactorings are performed using QVT, a transformation language similar to ATL, (iii) pre- and post-conditions are also implemented in OCL, and (iv) Refactory also does not identify bad-smells. The main differences are: (i) Refactory does not care about synchronizing the metamodel after applying refactorings; and (ii) Refactory does not use the KDM metamodel to apply refactorings.

Another related tool is MOOSE [7, 29]. MOOSE accepts as input several types of programming language (Smalltalk, Java, C++, etc.) and then MOOSE transforms such languages into a meta-model called FAMIX [29]. After performing this transformation, refactorings can be applied. MOOSE uses the Refactoring Browser [26]. Therefore, refactorings can be applied to instances of FAMIX; then automatically replicated in the Smalltalk source code. In the same way that KDM-RE, MOOSE, allows the application of language-independent refactorings, KDM-RE and MOOSE allow the application of refactorings through diagrams.

A recent research suggest that quality metrics do not show a clear relationship with refactoring [3]. In other words, quality metrics might suggest classes as good candidates to be refactored that are generally not involved in developer's refactoring operations. The authors reached such conclusion by mining the evolution history of three Java open source projects. They investigate whether refactoring activities occur on code components for which certain indications - such as quality metrics or the presence of smells as detected by tools - suggest there might be need for refactoring operations. They analyzed 12.922 refactoring operations - where 42% (5425) are performed by developers on code smells. Nonetheless, of these 42% only 7% (933) actually remove the code smell from the affected class and 895 are attributable to only four code smells (i.e., Blob, Long Method, Spaghetti Code, and Feature Envy). Thus, not all code smells are likely to trigger refactoring activities. Summing up, the results suggest that refactoring actions are not a direct consequence of worrisome metric profiles or of the presence of code smells, but rather driven by a general need for improving maintainability. Furthermore, the authors argue that refactoring recommendation tools should not only base their suggestions on code characteristics, but they should consider the developer's point-of-view in order to propose meaningful suggestions of classes to be refactored.

Another research conducted a longitudinal study to verify whether and to what extent refactoring improve software structural quality [6]. The authors analyzed how often 2.635 refactorings, which covered 11 of the most-common types, affected the density of 5 types of code smells along the version histories of 25 open source projects. Their study showed that 95.1% of the refactorings did not reduce code smells, only 2.24% removed some of them and 2.66% introduced new ones. Thus, according to their study, refactorings do not affect code smell as good as what has been reported. Both the researches [3, 6] share some common results and point that refactorings are applied indiscriminately. We emphasize that KDM-RE only facilitates the refactoring process of KDM instances. However, it is the responsibility of developers to identify which refactorings should be applied.

6 CONCLUDING REMARKS

We presented a tool, called KDM-RE, developed to assist software engineers during system modernization efforts. More

specifically, KDM-RE allows software engineers to apply refactorings to KDM instances. We carried out an experiment to evaluate the advantages of using KDM-RE to refactor systems through their KDM instances. During the experiment, we applied refactorings to seven real-world systems using our refactoring tool. The main objective was to verify whether applying a set of refactorings to remove bad-smells in KDM instances increases the quality of the systems. In the context of our experiment, four quality attributes were used as proxy for quality: reusability, flexibility, understandability, and effectiveness.

The results of our experiment would seem to suggest that our tool can help software engineers to apply refactorings that improve the quality of software systems. More specifically, using our tool to refactor the chosen software systems results in several improvements: after applying our proposed refactorings to the seven systems, all systems improved: reusability (12.77%), flexibility (6.37%), understandability (17.79%), and effectiveness (5.29%). According to the results of the statistical tests we performed, the first three attributes (i.e., reusability, flexibility, and understandability) increased significantly. Although there was also an increase in effectiveness, the statistical tests indicated that it was not significant.

As future work, we intend to compare KDM-RE with other similar model-based refactoring tools. Furthermore, as short-term future work, given that refactoring tools should take the developer's feedback into account rather than only bad smells, we plan on extending KDM-RE so as to allow it to consider the developer's expertise during refactoring activities.

ACKNOWLEDGMENTS

We would like to thank the financial support of FAPESP, process number 2016/03104-0. Rafael Durelli also would like to thank the financial support of FAPEMIG.

REFERENCES

[1] Thorsten Arendt and Gabriele Taentzer. 2013. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering* 20, 2 (2013), 141–184.
[2] J. Bansiya and C.G. Davis. 2002. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering* 28, 1 (2002), 4–17.
[3] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107, 2 (2015), 1 – 14.
[4] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka. 2006. Automated Impact Analysis of UML Models. *Journal of System Software.* 79, 3 (2006), 339–352.
[5] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. 2014. Modisco: A model driven reverse engineering framework. *Information and Software Technology* 56, 8 (2014), 1012–1032.
[6] D. Cedrim, L. Sousa, A. Garcia, and R. Gheyi. 2016. Does refactoring improve software structural quality? A longitudinal study of 25 projects. In *XXXI Brazilian Symposium on Software Engineering.*
[7] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. 2000. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Second International Symposium on Constructing Software Engineering Tools.*
[8] R.S. Durelli, B.M. Santos, R.R. Honda, M. E. Delamaro, and V.V. de Camargo. 2014. KDM-RE: A Model-Driven Refactoring Tool for KDM. In *Workshop on Software Visualization, Maintenance, and Evolution (VEM), 2014.*
[9] Rafael Serapilha Durelli, Daniel Martín Santibáñez, M. E. Delamaro, and Valter Vieira de Camargo. 2014. Towards a refactoring catalogue for knowledge discovery metamodel. In *15th International Conference on Information Reuse and Integration (IRI).*
[10] Eclipse. 2017. Eclipse Modeling Framework (EMF). https://eclipse.org/modeling/emf/. (2017).
[11] Eclipse. 2017. MoDisco. http://eclipse.org/MoDisco/. (2017).
[12] Eclipse. 2017. Papyrus. https://eclipse.org/papyrus/. (2017).
[13] A. Egyed, E. Letier, and A. Finkelstein. 2008. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *23rd IEEE/ACM International Conference on Automated Software Engineering.*
[14] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.
[15] Adam C. Jensen and Betty H.C. Cheng. 2010. On the Use of Genetic Programming for Automated Refactoring and the Introduction of Design Patterns. In *12th Annual Conference on Genetic and Evolutionary Computation.*
[16] Marouane Kessentini, Stéphane Vaucher, and Houari Sahraoui. 2010. Deviance from Perfection is a Better Criterion Than Closeness to Evil when Identifying Risky Code. In *IEEE/ACM International Conference on Automated Software Engineering.*
[17] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, and P. Van Gorp. 2014. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming* 85, 3 (2014), 5 – 40.
[18] Tom Mens and Pieter Van Gorp. 2006. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science* 152, 3 (2006), 125–142.
[19] Mohammed Misbhauddin and Mohammad Alshayeb. 2015. UML model refactoring: a systematic literature review. *Empirical Software Engineering* 20, 1 (2015), 206–251.
[20] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. 2016. On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach. *Empirical Software Engineering* 21, 6 (2016), 2503–2545.
[21] OMG. 2017. Architecture-Driven Modernization. http://adm.omg.org/. (2017).
[22] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks.* Ph.D. Dissertation. University of Wisconsin.
[23] Ricardo Perez-Castillo, Ignacio García-RodrÃŋguez de Guzmán, Orlando Ávila-García, and Mario Piattini. 2009. On the Use of ADM to Contextualize Data on Legacy Source Code for Software Modernization. In *16th Working Conference on Reverse Engineering.*
[24] Roger Pressman. 2010. *Software Engineering: A Practitioner's Approach* (7 ed.). McGraw-Hill.
[25] Jan Reimann, Mirko Seifert, and Uwe Aßmann. 2010. Role-based generic model refactoring. In *Model Driven Engineering Languages and Systems.*
[26] Don Roberts, John Brant, and Ralph Johnson. 1997. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems* 3, 4 (1997), 253–263.
[27] Ramzi Ben Salem, Reyes Grangel, and Jean-Pierre Bourey. 2008. A comparison of model transformation tools: Application for Transforming GRAI Extended Actigrams into UML Activity Diagrams. *Computers in Industry* 59, 7 (2008), 682 – 693.
[28] M Staroń and L Kuźniarz. 2004. Implementing UML Model Transformations for MDA. In *11th Nordic Workshop on Programming and Software Development and Tools.*
[29] Sander Tichelaar, Stkphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. 2000. A meta-model for language-independent refactoring. In *International Symposium on Principles of Software Evolution.*
[30] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. 2003. Towards Automating Source-Consistent UML Refactorings. 40, 1 (2003), 144–158.
[31] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering: an introduction.* Kluwer Academic Publishers.