Towards a Refactoring Catalogue for Knowledge Discovery Metamodel

Rafael S. Durelli[†], Daniel S. M. Santibáñez[*], Márcio E. Delamaro[†] and Valter Vieira de Camargo[*]
[†]Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: {rdurelli, delamaro}@icmc.usp.br
[*]Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil
Email: {daniel.santibanez,valter}@dc.ufscar.br

***Abstract*—Refactorings are a well known technique that assist developers in reformulating the overall structure of applications aiming to improve internal quality attributes while preserving their original behavior. One of the most conventional uses of refactorings are in reengineering processes, whose goal is to change the structure of legacy systems aiming to solve previously identified structural problems. Architecture-Driven Modernization (ADM) is the new generation of reengineering processes; relying just on models, rather than source code, as the main artifacts along the process. However, although ADM provides the general concepts for conducting model-driven modernizations, it does not provide instructions on how to create or apply refactorings in the Knowledge Discovery Metamodel (KDM) metamodel. This leads developers to create their own refactoring solutions, which are very hard to be reused in other contexts. One of the most well known and useful refactoring catalogue is the Fowler's one, but it was primarily proposed for source-code level. In order to fill this gap, in this paper we present a model-oriented version of the Fowler's Catalogue, so that it can be applied to KDM metamodel. In this paper we have focused on four categories of refactorings: (*i*) renaming, (*ii*) moving features between objects, (*iii*) organizing data, and (*iv*) dealing with generalization. We have also developed an environment to support the application of our catalogue. To evaluate our solution we conducted an experiment using eight open source Java application. The results showed that our catalogue can be used to improve the cohesion and coupling of the legacy system.**

***Index Terms*—Refactoring, KDM, ADM, Empirical Study**

I. Introduction

Empirical studies show that refactoring can improve maintainability and reusability of systems [1]. Not only existing research which suggests that refactoring is useful, but it also suggests that refactoring is a frequent practice [2].

In a parallel and related research line, Object Management Group has employed a lot of effort to define standards in the refactoring process, creating the concept of ADM. ADM is the next generation of software reengineering, relying on standard models to perform the whole process. ADM follows the Model-Driven Development (MDD) [3] guidelines and comprises two major steps. Firstly a reverse engineering is performed starting from the source code and a model instance is created. Next successive transformations are applied to this model up to reach a good abstraction level in model called KDM. Upon this model, several refactorings, and optimizations can be performed in order to solve problems found in the legacy system. Secondly a forward engineering is carried out and the source code of the modernized target system is generated. According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing information technology architectures. The idea behind the standard KDM is that the community starts to create parsers from different languages to KDM. As a result everything that takes KDM as input can be considered platform and language-independent. For example, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages. The current version of the KDM is 1.3 and it is being adopted by ISO as ISO/IEC 19506 [3].

In the area of object-oriented programming, refactorings are the technique for improving the structure of legacy system without changing its external behavior [2]. Nowadays it is evident that refactorings are useful to improve the quality of source code, and thus, to increase its maintainability. However, although ADM, and mainly KDM, had been proposed to support the modernization of legacy systems, up to this moment there are no proposals of refactoring catalogues for KDM. Therefore, software modernizers/reengineers need to develop your own solutions to transform source KDM instances in target ones. Usually these solutions are proprietary and very difficult to reuse. Besides, available object-oriented refactoring catalogues can not be reusable as they are, because they have been created to source-code refactoring. This forces the software engineer to refactor a legacy system without any kind of dedicated support as using the KDM model.

As refactoring a legacy system can be very complex, manual modifications without any catalogue may lead to unwanted side-effects and result in a tedious and error-prove refactoring process. Therefore, we claim that working out a catalogue of refactoring to the KDM specification is indispensable because software engineers can reduce time and effort during the refactoring of legacy systems once we are using techniques of MDD [4]. Furthermore, we also argue that devising a catalogue of refactoring by means of KDM specification makes this catalogue be both language-independent and standardized [5].

We believe there are two main hurdles to be overcome so that refactoring techniques can be used in the KDM model in an effective and widespread way. The first hurdle is the lack a catalogue of well known refactorings based on the KDM model. Also, software modernizers would greatly benefit from the possibility to follow a catalogue of refactoring, in practice they mostly rely on experience or intuition because of the lack of approaches providing a catalogue for KDM. In order to address this first hurdle we present a dedicated refactoring catalogue for the KDM metamodel which is based on the catalogue proposed by Fowler [2]. We chose this catalog once it contains well known, basic and fine-grained refactorings. This allows that larger refactorings can be applied when combining a sequence of them, i.e., a chain of refactorings.

A second hurdle is the absence of an Integrated Development Environments (IDEs) to lead engineers to automatically apply refactorings using the KDM model as such exist in others object-oriented languages. The catalogue presented by Fowler et al. [2] provided a basis on which developers could rely to build tool support for object-oriented refactoring: similar catalogue for the KDM models are likely to bring similar benefits to assist software modernizers during the modernization process. Therefore, to overcome the second hurdle we devised a Eclipse plug-in named Modernization-Integrated Environment (MIE), which is an environment that implements all refactoring available in the catalogue herein. The novelty of this environment is not the supporting technologies and tools, but rather its catalogue of refactoring based on KDM.

Moreover, in order to provide some evidence of the our dedicated refactoring catalogue for KDM we performed an experiment using eight open source Java application. More specifically, we conducted a reverse engineering in order to get the KDM model of these eight open source Java application. Then, we applied three different refactorings in their KDM models. Experimental results show that the our dedicated refactoring catalogue improved the legacy system's cohesion.

Therefore, the main contributions of this paper are threefold: (*i*) we show a dedicated refactoring catalogue for KDM; (*ii*) we demonstrate the feasibility of our catalogue by implementing it as an Eclipse plug-in; *(iii)* we show the results of an experiment by applying some refactorings to eight open source programs and their KDM models.

This paper is structured as follows: in Section II ADM and KDM are explained; in Section III, the catalogue of refactoring for KDM is shown; in Section IV, a Eclipse plug-in to support the catalogue is described; Section V summarizes the experimental results; in Section VI, there are related works and Section VII suggests future work and makes concluding remarks.

II. Background

A. ADM and KDM

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 MDD [6, p. 60].

To perform a system modernization, ADM introduces several modernization standards: Abstract Syntax Tree Metamodel (ASTM), Knowledge Discovery Metamodel (KDM), Structured Metrics Metamodel (SMM), System Assurance & Evidence, Software Quality and Business Architecture Standards. These standards collectively provide or can provide the universe of metadata that defines existing software environments. However, here we focus on KDM because it is the key cornerstone of ADM and the main ideas of our research. The goal of the KDM standard is to define a metamodel to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, etc.). The metamodel of the KDM standard provides a comprehensive high-level view of the behavior, 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 generate new code in a top-down manner, 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.

KDM specification owns some KDM domain, each domain defines an architectural viewpoint. In order to define the catalogue of refactoring for the KDM we need to focus just on the Program Element Layer - more specifically in the Code Package, which represents the code elements of a program (classes, fields and methods) and their associations. Therefore, it is important to dig a little deeper in the Code Package. The Code Package consists of 24 classes and contains all the abstract elements for modeling the static structure of the source code. In Table I is depicted some of them. This table identifies KDM metaclasses possessing similar characteristics to the static structure of the source code. Some metaclasses can be direct mapped, such as Class from object-oriented language, which can be easily mapped to the ClassUnit metaclass from KDM.

Table I
METACLASSES FOR MODELING THE STATIC STRUCTURE OF THE SOURCE-CODE

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

III. Refactoring Catalogue for Knowledge Discovery Metamodel

In this section we describe the catalogue of refactoring for the KDM herein proposed. To create this catalogue we adapted some fine-grained refactorings proposed by Fowler [2]. As stated before we chose the Fowler's refactorings once them are well known, basic and fine-grained refactorings. It is worth highlighting that for drafting the catalogue of refactorings in this paper, we also used a format similar to Fowler [2].

The catalogue is structured in four groups as can be seen in the Table II, which contains 17 refactorings followed by

Table II
REFACTORING CATALOGUE FOR KNOWLEDGE DISCOVERY METAMODEL

N	Name of the Refactoring	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.

a short description. Due to space limitations in the followings subsections is described only three refactorings: **Extract ClassUnit**, **Replace data value with Object** and **Pull Up MethodUnit**. In order to highlight how these refactorings can be applied into KDM we also show two algorithms. These algorithms can assist other software modernizers to create news refactorings once these refactorings show explicitly how to handle the KDM metamodel. In the followings subsections a description of these refactorings are provided.

A. *Extract ClassUnit*

Input: A source ClassUnit to extract responsibilities, a name of the new class, instances of the meta-class that represent StorableUnit to be moved and instances of the meta-class that represent MethodUnits to be moved.

Summary: You have one ClassUnit doing work that should be done by two.

Solution: Create a new ClassUnit and move the relevant StorableUnit and MethodUnits from the old ClassUnit into the new ClassUnit.

Parameters:

- A source ClassUnit to extract responsibilities.
- The name of the new ClassUnit.
- Instances of the meta-class that represent StorableUnits to be moved.
- Instances of the meta-class that represent MethodUnits to be moved.

Refactoring Guidelines:

- Split the responsibilities of the class.:
  - Identify the StorableUnits that should not be in source ClassUnit.
  - Identify the MethodUnits that should not be in source ClassUnit.
- Create a new instance of ClassUnit to express the split-off responsibilities.
- Use **Move StorableUnit** on each StorableUnit you wish to move.
- Use **Move MethodUnit** on each MethodUnit you wish to move.

Algorithm 1: Extract ClassUnit

```
Input: String newName, ClassUnit class, Package p,
       StorableUnit[] fields, MethodUnit[] methods
begin
    ClassUnit extracted =
    CodeFactory.eINSTANCE.createClassUnit() ❶;
    if extracted != null then
        extracted.setName(newName);
        extracted.getSource.add(SourceFactory.
        eINSTANCE.createSourceRef());
        p.getCodeElement().add(extracted);
    else
        else return null
    end
    StorableUnit link =
    CodeFactory.eINSTANCE.createStorableUnit() ❷;
    if link != null then
        link.setName(extracted.getName().toLowerCase());

        link.getAttribute().add(KdmFactory.eINSTANCE.
        createAttribute());
        link.getSource().add(SourceFactory.eINSTANCE.
        createSourceRef());
        extracted.getCodeElement().add(link);
    else
        else return null
    end
    foreach f in fields do
        extracted.getCodeElement().add(f) ❸;
    end
    foreach m in methods do
        extracted.getCodeElement().add(m) ❹;
    end
    return extracted
end
```

To illustrate how the refactoring **Extract ClassUnit** can be implemented in the KDM model, consider the chunk of pseudo code depicted in Algorithm 1. Before the line 2 be executed one must give the following inputs: (*i*) a name to set the new ClassUnit, (*ii*) a source ClassUnit to extract either StorableUnit or MethodUnit, (*iii*) a Package to put the new ClassUnit, and (*iv*) a set of StorableUnit and MethodUnit. In line 2 an instance of ClassUnit is created. This ClassUnit is created by means of the Abstract Factory Pattern ❶. If the condition in line 3 evaluates to true, the statements in lines 4, 5 and 6 are executed. During their execution the name of the new ClassUnit is set and in line 6 the new ClassUnit is added to the instance of Package. In line 10 a StorableUnit is created ❷ - it represents a link from the old ClassUnit to the new ClassUnit. If the condition in line 11 evaluates to true, the statements in lines 12, 13, 14 and 15 are executed. In line 12 is set the name of the StorableUnit. Line 15 add the created StorableUnit to the created ClassUnit. Lines 19, 20 and 21 illustrate a loop to move all StorableUnit to the new ClassUnit ❸. Similarly, lines 22, 23 and 24 depict another loop to move all MethodUnit to the new ClassUnit ❹.

B. *Replace Data Value With Object*

Input: Instance of the meta-class that represent the StorableUnit that needs additional data or behavior.

Summary: You have a StorableUnit that needs additional data or behavior.

Solution: Turn the StorableUnit into a ClassUnit.

Parameters:

- Instance of the meta-class that represent the StorableUnit that needs additional data or behavior.

Refactoring Guidelines:

- Create a ClassUnit for the StorableUnit.
- Create a StorableUnit of the same type as the value in the new ClassUnit.
- Create an instance of MethodUnit to represent the operation get that takes the StorableUnit as an argument.
- Change the type of the StorableUnit in the source ClassUnit to the new ClassUnit.
- Change the getter in the source ClassUnit to call the getter in the new ClassUnit.

C. *Pull Up MethodUnit*

Input: Instances of subclasses that own in common the superclass.

Summary: A set of subclasses that have at least one MethodUnit in common.

Solution: Move the commons MethodUnits to the superclass.

Precondition: Identify the commons MethodUnits.

Parameters:

- Instances of subclasses that own in common the superclass.

Refactoring Steps:

- For each instance that represent the subclasses identify if there is some MethodUnit in common between these subclasses.
- For each identified MethodUnit applies the refactoring **Move Method**. As parameter to this refactoring it is necessary the identified MethodUnit and the an instance of the ClassUnit that represents the superclass to move it.

In Algorithm 2 is illustrated how the refactoring **Pull Up MethodUnit** can be implemented in the KDM model. Before to start the refactoring **Pull Up MethodUnit** the line 2 must be executed ❶. The statement in this line inspects all MethodUnits in a set of subclasses to ensure they are identical. If the condition in line 3 evaluates to true, the loop in lines 4, 5 and 6 are executed. In this loop all identical MethodUnit are moved to an instance of ClassUnit that represent the superclass ❷. In line 10 all elements of the superclass are obtained ❸. Then two loop are execute in lines 11 and 12, respectively. The inner loop (line 12) verify if two MethodUnits are equals. If the condition in line 13 evaluates to true, then a cast is made in line 14 and the contained MethodUnit is removed in line 15.

Algorithm 2: Pull Up MethodUnit

```
   Input: ClassUnit[] subClasses, ClassUnit superC
 1 begin
 2     MethodUnit[] commonMethods =
       identifyCommonMethodUnit(subClasses) ❶;
 3     if commonMethods != null then
 4         foreach mU in commonMethods do
 5             superC.getCodeElement().add(mU) ❷ ;
 6         end
 7     else
 8         else return null
 9     end
10     EList[] elements = superC.getCodeElement() ❸ ;
11     foreach c in elements do
12         foreach c2 in elements do
13             if (c instanceof MethodUnit) &&
               (c.getName().equals(c2.getName())) then
14                 MethodUnit mRve = (MethodUnit) c2;
15                 superC.getCodeElement().remove(mRve);
16             end
17         end
18     end
19 end
```

IV. Proof-Of-Concept Implementation

We devised a Eclipse plug-in named Modernization-Integrated Environment (MIE) which is split in three layers, as follows: (*i*) Core Framework, (*ii*) Tool Core, and (*iii*) Graphical User Interface (GUI). This plugin was devised on the top of the Eclipse Platform; The first layer we used both Java and Groovy as programming language. Moreover, the Core Framework layer contains a set of Eclipse plug-ins on which our environment is based on, such as MoDisco and Eclipse Modeling Framework (EMF)[1]. We used MoDisco[2] once it is

[1] http://www.eclipse.org/modeling/emf/

[2] http://www.eclipse.org/MoDisco/

an extensible framework to develop model-driven tools to support use-cases of existing software modernization and provides an Application Programming Interface - (API) to easily access the KDM model. Also, EMF was used to load and navigate KDM models that were generated with MoDisco. The second layer, the Tool Core, is where all refactorings provided by our environment were implemented. It works intensively with KDM models, which are XML files. Therefore, we use Groovy to handle those types of files because of the simplicity of its syntax and fully integrated with Java. Finally, the third layer is the Graphical User Interface (GUI) that consists of a set of SWT windows with several options to perform the refactorings based on the KDM model.

V. Experimental Study

This section describes the experiment used to gauge the catalogue of refactoring for KDM. Moreover, this experiment also evaluate the devised Eclipse plug-in, which was earlier described. Specifically, we investigate the following research question:

RQ$_1$: How much of the program's maintainability and understandability (high cohesion) can be affected by applying a set of refactorings in the KDM model?

A. Goal Definition

We use the organization proposed by the Goal/Question/-Metric (GQM) paradigm, it describes experimental goals in five parts, as follows: (*i*) **object of study:** the object of study is our catalogue of refactoring adapted for KDM; (*ii*) **purpose:** the purpose of this experiment is to evaluate the catalogue of refactoring adapted for KDM; (*iii*)**perspective:** this experiment is run from the standpoint of a researcher; (*iv*) **quality focus:** the primary effect under investigation is the improvement in program's maintainability and understandability (high cohesion) after applying the refactorings; (*v*) **context:** this experiment was carried out using Eclipse 4.3.2 on a 2.5 GHz Intel Core i5 with 8GB of physical memory running Mac OS X 10.9.2. Our experiment can be defined as: **Analyze** the catalogue of refactoring adapted for KDM model, **for the purpose of** evaluation, **with respect to** improvement in program cohesion, **from the point of view of** the researcher, **in the context of** heterogeneous subject programs.

B. Hypothesis Formulation

Our research question (**RQ**$_1$) was formalized into hypotheses so that statistical tests can be performed.

Null hypothesis, H$_0$: there is no difference in cohesion before and after to apply a set of refactoring into the KDM model (measured in terms of the metric CAMC and SCC) which can be formalized as:

H$_0$**:** $\mu_{CAMC_{Bf}} = \mu_{CAMC_{Af}}$ **and** $\mu_{SCC_{Bf}} = \mu_{SCC_{Af}}$

Alternative hypothesis, H$_1$: there is a significant difference in cohesion before and after to apply a set of refactoring into the KDM model (measured in terms of the metric CAMC and SCC) which can be formalized as:

H$_1$**:** $\mu_{CAMC_{Bf}} \neq \mu_{CAMC_{Af}}$ **and** $\mu_{SCC_{Bf}} \neq \mu_{SCC_{Af}}$

C. Experimental Design

For our evaluation, we need a set of sample KDM models to apply the catalogue of refactoring. However, due to the scarcity of complete KDM models in the public domain, we adopted a reverse engineering approach and generated KDM models from eight open source Java projects by using MoDisco. During the selection of these programs we focused on covering a broad class of applications. In addition, several of the subject programs have been studied elsewhere, making this study comparable with earlier studies. Table III shows the subject KDM models along with some measures of their size. Notice that these measures were obtained automatically by MoDisco and Eclipse Metrics 1.3.6 [3].

Table III
KDM Models used in the Evaluation

ID	**Program**	**KDM Model**		
		ClassUnit	**StorableUnit**	**MethodUnit**
1	org.gadberry.jexel	75	199	240
2	Jester	14	10	59
3	apache.commons.cli	99	707	621
4	apache.commons.io	357	643	2453
5	JUnit	1041	712	2552
6	org.jaxen	353	440	2063
7	org.snmp4j	329	1146	2340
8	org.dom4j	469	653	3032
Total		2737	4510	13360

We selected three refactorings for our evaluation: **Extract ClassUnit**, **Push Down MethodUnit**, and **Pull up MethodUnit**. We applied each of the three refactorings to every possible location in each KDM model. It is worth to notice that all refactorings were applied completely automatically by means of our devised proof-of-concept tool. To deal with refactorings that go into infinite loops, we set three minutes timeout interval. More specifically, we applied the **Extract ClassUnit** to every class that had more than 300 LOC (Line of Code); we applied the **Push Down MethodUnit** to every method of a class that had a subclass that was not from a library using every such subclass as the target of the *push-down*; and we applied the refactoring **Pull up MethodUnit** to every method of a class that had a superclass that was not from a library, using every such superclass as the target of the *pull-up*. Then after applied all refactorings we counted whether them were successful, i.e., if the intended refactoring could be performed, and how many constraints were generated on the model and on the code side after to apply the refactorings. We also measured both software quality metrics Cohesion Amongst the Methods of a Class (CAMC) and Similarity-based Class Cohesion (SCC)[4] before applying the refactoring on the KDM models and after applying the refactoring on the KDM models. Notice that in this case we actually measured these metrics in the code instead of the KDM model. This was possible as our proof-of-concept tool provides support for the generating of the code after one finishes to apply the refactorings.

[3] http://metrics.sourceforge.net/

[4] CAMC and SCC both are defined as high-level design quality metrics, and an increase in their value means an improvement in program cohesion.

Table IV
RESULTS OF APPLICATION PER PROJECT AND PER REFACTORING

ID	Extract ClassUnit							Push Down MethodUnit							Pull Up MethodUnit						
	Exec.	Succ.	Cons.	CAMC		SCC		Exec	Succ.	Cons.	CAMC		SCC		Exec	Succ.	Cons.	CAMC		SCC	
				Bf	Af	Bf	Af				Bf	Af	Bf	Af				Bf	Af	Bf	Af
1	75	43	32	0.133	0.189	0.078	0.092	350	245	105	0.133	0.164	0.078	0.094	350	311	39	0.133	0.177	0.078	0.098
2	14	7	7	0.168	0.196	0.084	0.096	267	213	54	0.168	0.171	0.084	0.087	267	213	54	0.168	0.178	0.084	0.097
3	99	48	51	0.163	0.176	0.067	0.081	159	130	29	0.163	0.152	0.067	0.073	159	130	29	0.163	0.187	0.067	0.085
4	357	267	90	0.175	0.185	0.088	0.094	444	124	320	0.175	0.159	0.088	0.098	444	124	320	0.175	0.187	0.088	0.089
5	1041	705	336	0.183	0.199	0.063	0.089	468	284	184	0.183	0.1773	0.063	0.076	468	284	184	0.183	0.194	0.063	0.079
6	353	156	197	0.193	0.167	0.087	0.073	1411	841	570	0.193	0.168	0.087	0.079	1411	851	570	0.193	0.198	0.087	0.087
7	329	298	31	0.128	0.198	0.092	0.099	678	489	189	0.128	0.158	0.092	0.095	678	489	189	0.128	0.165	0.092	0.099
8	469	389	80	0.17	0.159	0.085	0.097	398	368	30	0.17	0.15	0.085	0.091	398	368	30	0.17	0.25	0.085	0.097

D. Results and Empirical Analysis for the Cohesion Values

This section presents the experimental results after to apply the refactorings **Extract ClassUnit**, **Push Down MethodUnit**, and **Pull up MethodUnit**. The results are depicted in Table IV. The columns Exec., Succ., and Cons. stand for Executed, Successful and Constraints, respectively. Columns Exec. shows the number of refactorings applied to the original KDM model. Columns Succ. presents the rate of successful refactoring really applied in the KDM model, otherwise Columns Cons. shows the rate of unsuccessful refactoring applied in the KDM model. As can be seen, success rates, constraints generated and changes induced vary widely for every refactoring. As stated before, we also measured some software quality metrics (CAMC and SCC) before and after applying all refactorings. Therefore, Columns CAMC and SCC are split into two cell, i.e., Bf and Af as Bf stand for Before and Af stand for After. As shown, the quality in terms of cohesion is in some case was gradually degraded. However, it is fairly evident that in some case the applied refactorings improved the cohesion, i.e., in some points we had positive impact on the design quality as shown in Table IV.

In Figure 1 is summarized the sampled data of the metrics CAMC and SCC before and after to apply the refactorings. These figure also provide an overview of the significant gain in cohesion that could be achieved by our catalogue of refactoring. Besides achieving gain in cohesion, in this figure it is fairly evident that for almost every refactoring the median after applying the refactorings of both metrics (CAMC and SCC) is greater than before to apply the refactorings. In addition, in these figures also can be observed that the interquartile ranges are reasonably similar (as shown by the lengths of the boxes), though the overall range of the data set is greater before to apply the refactorings (as shown by the distances between the ends of the two whiskers for each boxplot). Just the data set related to the metric CAMC for **Pull Up Method** shows one suspiciously far out values (outliers) which required a closer look.

In Table IV it is possible to point out by looking at columns Succ. that the rate of successful applied refactoring are better than the the rate of unsuccessful. In order to be aware of some unsuccessful refactorings, we manually inspected some refactorings to find the reason why some refactorings could not be applied to the KDM model during the refactoring process. Then we found out that the problem mostly happened when a hierarchy structure of the KDM file was changed radically.

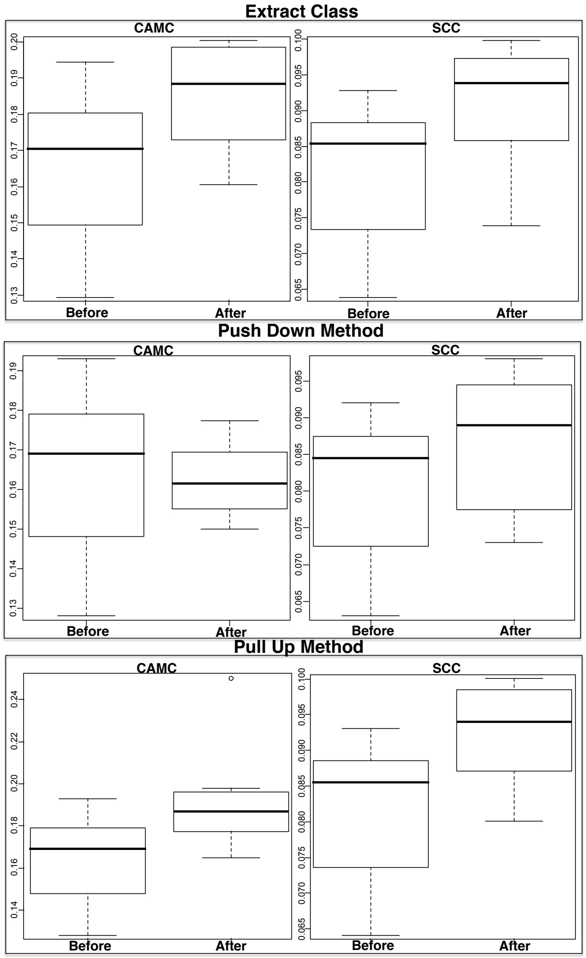

Figure 1. Comparative boxplots for the refactorings.

Since some statistical tests only apply if the population follows a normal distribution, before choosing a statistical test we examined whether our sample data departs from linearity. We use Q-Q plots as shown in Figure 2. In these figure one can see that most of the data depart from linearity, indicating normality of the samples. Therefore, we could apply the t-

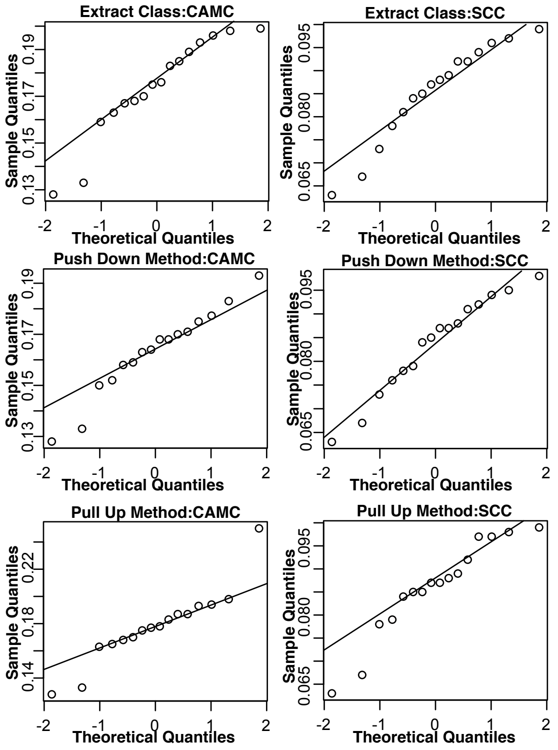

Figure 2. Normal probability plots.

test. In our context, to apply the t-test for CAMC and SCC: (1) the difference between the cohesion values before and after the change has to be calculated, (2) the means and standard deviation values of these differences have to be obtained, and (3) the t-value has to be calculated. We applied the typical significance threshold ($\alpha = 0.05$) to decide whether the differences between the cohesion values were significant.

The results in Tables IV and V lead us to the following observations about the refactoring **ExtractClassUnit**.

The results before and after applying the refactoring **ExtractClassUnit** demonstrate that the cohesion values for the metric CAMC were significantly different. As can be seen in Table V the means before and after applying the refactoring **ExtractClassUnit** were 0.164125 and 0.183625, respectively. The corresponding critical T value was -2.6139, the standard deviation for this refactoring was 0.0211, DOF $= 7$, two-tailed p-value $=> 0.0347$, with 95% confidence intervals (upper $= 0.1817$ and lower$= 0.1465$). Interpreting these data we can remark that the difference in regarding to the metric CAMC for the refactoring **ExtractClassUnit** are significantly different. Therefore, we can conclude with 95% confidence (or a chance of error of 5%) that the refactoring **ExtractClassUnit** helped to increase the cohesion amongst the methods of a class of the evaluated systems. Regarding the results of the metric SCC after applying the refactoring **ExtractClassUnit** we could validate that the evaluated systems had significant change. As can be seen in Table V the T value was -2.5616, the standard deviation for this refactoring was 0.0106, DOF $= 7$, two-tailed p-value $= 0.0375$, with 95% confidence intervals (upper $= 0.0894$ and lower$= 0.0716$). Interpreting these data we can observe that the difference in regarding to the metric SCC for the refactoring **ExtractClassUnit** are significantly different. Thus, we can conclude with 95% confidence (or a chance of error of 5%) that the refactoring **ExtractClassUnit** improved in some way the similarity-based class cohesion of the evaluated systems.

Similarly, the results in Tables IV and V lead us to the following observations about the refactoring **PushDownMethodUnit**. The results before and after applying the refactoring **PushDownMethodUnit** demonstrate that the cohesion values for the metric CAMC were not significantly different. As can be seen in Table V the means before and after applying the refactoring **PushDownMethodUnit** were 0.164125 and 0.1624125, respectively. The corresponding critical T value was 0.2862, the standard deviation for this refactoring was 0.0168, DOF $= 7$, two-tailed p-value $= 0.783$, with 95% confidence intervals (upper $= 0.1781$ and lower$= 0.1501$). Interpreting these data according to the t-test we can remark that metric CAMC for the refactoring **PushDownMethodUnit** are not significantly different. Therefore, it was possible to conclude with 95% confidence (or a chance of error of 5%) that the refactoring **PushDownMethodUnit** did not improve the cohesion amongst the methods of a class of the evaluated systems. As for the metric SCC the refactoring **PushDownMethodUnit** demonstrate that the cohesion values are also not significantly different. Although, in Table IV one can point that after applying the refactoring **PushDownMethodUnit** all the evaluated system had been improved in regarding the metric SCC (see column SCC cell Af) - after perform the t-test we could conclude with 95% confidence (or a chance of error of 5%) that the refactoring **PushDownMethodUnit** did not raise the similarity-based class cohesion of the evaluated systems.

Finally, the results in Tables IV and V lead us to the following observations about the refactoring **PullUpMethodUnit**. Similarly as the refactoring **PushDownMethodUnit** the refactoring **PullUpMethodUnit** demonstrate that the cohesion values for the metric CAMC were not significantly different. As can be seen in Table V the means before and after applying the refactoring **PullUpMethodUnit** were 0.164125 and 0.192, respectively. The corresponding critical T value was -0.1945, the standard deviation for this refactoring was 0.4057, DOF $= 7$, two-tailed p-value $= 0.8513$, with 95% confidence intervals (upper $= 0.5033$ and lower$= -0.1751$). Although it is fairly evident that the refactoring **PullUpMethodUnit** improved in some way the evaluated systems (see columns CAMC cell Af) we cannot prove statistically that it really improved the evaluated systems. Thus, statistically we can remark with 95% confidence (or a chance of error of 5%) that the refactoring **PullUpMethodUnit** did not improve the cohesion amongst the methods of a class of the evaluated systems. Regarding the metric SCC it was not possible to find a significant difference between the cohesion before to apply the refactoring **PullUpMethodUnit** versus the cohesion after to apply the refactoring **PullUpMethodUnit**. In Table V

Table V
T-TEST RESULTS.

T-TEST	**Extract ClassUnit**				**Push Down MethodUnit**				**Pull Up MethodUnit**			
	CAMC		SCC		CAMC		SCC		CAMC		SCC	
	MEAN				MEAN				MEAN			
	0.164125	0.183625	0.0805	0.090125	0.164125	0.1624125	0.0805	0.086625	0.164125	0.192	0.0805	0.091375
	t-value = -2.6139		t-value = -2.5616		t-value = 0.2862		t-value = -1.7083		t-value = -0.1945		t-value = -0.1949	
	SD = 0.0211		SD = 0.0106		SD = 0.0168		SD = 0.0101		SD = 0.4057		SD = 0.1567	

the means before and after applying applying the refactoring **PullUpMethodUnit** were 0.0805 and 0.091375, respectively. The corresponding critical T value was -0.1949, the standard deviation for this refactoring was 0.1567, DOF $= 7$, two-tailed p-value $= 0.851$, with 95% confidence intervals (upper $= 0.2115$ and lower$= -0.0505$). Therefore, by interpreting these data we can draw the conclusion with 95% confidence (or a chance of error of 5%) that the refactoring **PullUpMethodUnit** did not improve the similarity-based class cohesion of the evaluated systems.

E. Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. We argue that this is a problem that all software engineering research, since we have theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. A threat to construct validity stems from possible faults in the implementations of the techniques. With regard to our catalogue of refactoring, we mitigated this threat by running a carefully designed test set against several small example programs. Similarly, all the eight open source Java projects used in this experiment have been extensively used within academic circles, so we conjecture that this threat can be ruled out.

VI. RELATED WORK

In [7] an approach to specify generic refactorings is presented. Here, Moha et al. introduce a meta-metamodel (GenericMT), which enables the definition of generic refactorings on the Meta Object Facility (MOF) layer. This meta-metamodel contains structural commonalities of object-oriented models (e.g., classes, methods, attributes and parameters). Generic refactorings are then specified on top of the GenericMT.

Borger et al. [8] 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.

VII. CONCLUSIONS

We adapted the traditional notion of fine-grained refactoring to the KDM specification. As far as we know, this paper is the first one to define one catalogue of refactoring for KDM. We argue that devising a catalogue of refactoring for KDM makes it be both language-independent and standardized. To provide some evidence of our catalogue of refactoring, we conducted an experiment using eight open source Java application. More specifically, for these eight application we applied three different refactorings - **Extract ClassUnit**, **Push Down MethodUnit**, and **Pull up MethodUnit**. Experimental results show that the our catalogue of refactoring improved the legacy system.

As a future work, we aim at applying our catalogue in more case studies.From these case studies we can propose more fine-refactorings for KDM. We also aim to create macro-refactorings for the KDM in order to explore the role of configuration knowledge for achieving model-driven refactoring for KDM. Previously, we devised an approach for identifying concerns in KDM models [9]. Therefore, the next step is to create refactorings that take into account Crosscutting Frameworks [10].

ACKNOWLEDGMENTS

Rafael S. Durelli would like to thank the financial support provided by FAPESP, process number 2012/05168-4.

REFERENCES

[1] S. Demeyer, B. Du Bois, and J. Verelst, "Refactoring - Improving Coupling and Cohesion of Existing Code," *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004.

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

[3] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems," *Comput. Stand. Interfaces*, pp. 519–532, 2011.

[4] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, ser. FOSE 07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54.

[5] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini, "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*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 128–132. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.20

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

[7] N. Moha, V. Mahe, O. Barais, and J.-M. Jezequel, "Generic model refactorings," in *MoDELS*, pp. 628–643.

[8] M. Boger and T. Sturn, "Tools-support for model-driven software engineering," *Proceedings of Practical UML-Based Rigorous Development Methods*, pp. 490–496, 2002.

[9] D. Santibáñez, R. S. Durelli, B. Marinho, and V. V. de Camargo, "A Combined Approach for Concern Identification in KDM models," in *Latin-American Workshop on Aspect-Oriented Software Development*, ser. LAWASP '7, 2013.

[10] V. V. Camargo and P. C. Masiero, "An approach to design crosscutting framework families," in *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, 2008.