KnowDIME: An Infrastructure based on Architecture-Driven Modernization for Improving Legacy System

Rafael S. Durelli[1,3], **Márcio E. Delamaro**[1] **and Valter V. de Camargo**[2]

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

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

[3]RMoD Team, INRIA, Lille, France

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

Abstract.

1. Introduction

Architecture-Driven Modernization (ADM) which has been proposed by OMG (Object Management Group) and advocates conducting refactoring following the principles of MDA (Model-Driven Architecture) [**?**] has become a great candidate since it aims to promote *(i)* portability, *(ii)* interoperability and *(iii)* reusability. According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a vertical and horizontal standard metamodel that represents all elements of the existing information technology architectures. 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" packages which is widely used by our tool. It defines a set of metaclasses 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.

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 [ref] in the discipline of Object Oriented (OO) Programming 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. The most complete and influential was published by Fowler in [6] for refactoring of Java code. However, while software reengineers would greatly benefit from the possibility to assess different choices, 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 software refactoring to model 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 KDM-RE (**K**nowledge **D**iscovery **M**odel-**R**efactoring **E**nvironment). Firstly, this *plug-in* provides a model-driven refactoring based on KDM models. Secondly, it also 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. Finally, after the best alternate refactoring path is chosen by the engineer, a forward engineering is performed and the source code of the refactored target system is generated again. Notice that KDM-RE supports 16 refactorings that are heavily inspired by the refatorings given by Fowler [**?**]. These refactorings are split up into three groups. The first group named **Moving Features Between Objects** which consists of relatively simple refactorings such as moving and creating features. The second group called **Organizing Data** is a set of refactorings to be applied in order to make working with data easier. The third group is named **Dealing With Generalization** and it represents refactorings to mostly dealing with moving methods and fields around a hierarchy of inheritance. This paper is organized as followed: Section 2 provides information related to KDM-RE - Section 3 the architecture of KDM-RE is depicted - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. KDM-RE

In this section the **K**nowledge **D**iscovery **M**odel-**R**efactoring **E**nvironment (KDM-RE) is presented. In Figure 1 we depict the overall process of our technique which was adapted based on the horseshoe modernization model. It is split into three parts, they are: (*i*) Reverse Engineering, (*ii*) Refactorings and (*iii*) Forward Engineering. Furthermore, our process is divided into six levels (**Level-0** to **Level-1**), as can be seen in Figure 1.

2.1. Reverse Engineering

Firstly, the engineer starts the process in the **Level-0** by choosing an eclipse project which contains the source-code to realize the refactorings. After that, into **Level-1** the source-code need to be transformed into a Platform-Specific Model (PSM). This PSM is an instance of the source-code which represents all abstraction of the source-code. To realize this transformation we implemented a model extractor in Java. To implement this model extractor we carried out three activities: 1) definition of the Java grammar, 2) mapping of Java grammar elements to elements of Java Model, and 3) implementation of the model extractor. For the definition of the Java grammar we used the Xtext[1] framework. Using this framework we obtained automatically three artifacts: 1) a metamodel, 2) a textual editor and 3) a Java parser that allow us to recognize the elements of the Java grammar from code written in java. Figure 2 illustrates how KDM-RE manages to assist the engineer to get the instance of this PSM. Figure 2 (a) shows the eclipse project selected by the engineer - then after the engineer click in a popup menu named “Discovery Java Model” the Figure 2 (b) is shown an excerpt to indicate the correspondence with the legacy Java Model.

[1] https://www.eclipse.org/Xtext/

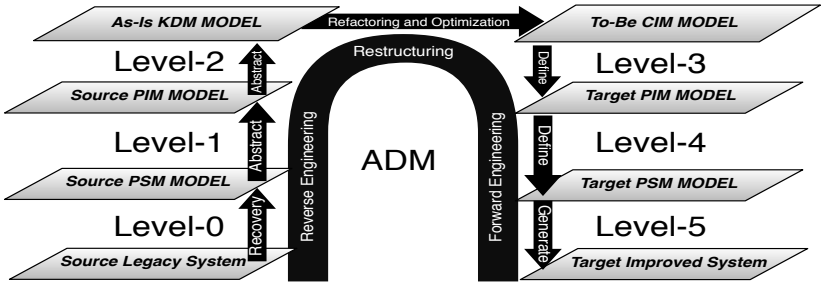

Figure 1. KDM-RE Process

After creating the PSM the next level (**Level-2**) consists in transforming the PSM to a Platform-Indented Model (PIM) which is based on the KDM. In this level the KDM-RE performs a set of M2M (Model-To-Model) transformation to get an instance of the KDM which represents the systems "AS-IS". These transformations are performed by ATL (ATL Transformation Language)[2]. Similarly, the KDM-RE also provides a popup menu named "Discovery KDM Model" which by clicking on it the engineer gets a instance of the KDM based on the earlier PSM. In Figure 2(c) shows how the Java Model is transformed to KDM.

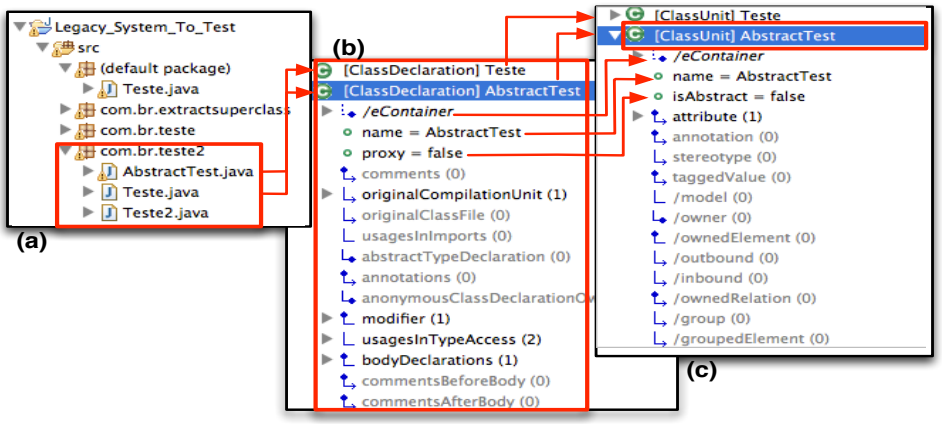

Figure 2. Process to Discovery Java and KDM Model

In Figure 3 we depicted the main window of our *plug-in*. For explanation purpose, we have identified main regions, i.e., ⓐ, ⓑ, ⓒ and ⓓ.

All refactorings provided by KDM-RE are based on the KDM model. In order to assist the refactorings we extended the KDM's model browser provided by MoDisco[3]. We added a popup menu named "Refactoring KDM" in this model browser, see Figure 3ⓐ. By using this menu the engineer can interact with the KDM model and choose which refactoring must be carried out in the model. In the region ⓐ can be seen all 16 refactorings that have been implemented in KDM-RE. The start of every refactoring is a engineer action in the Eclipse workbench. For illustration purposes only we drew rectangles to separate the refactorings into three groups. The black rectangle represents refactorings

[2]https://www.eclipse.org/atl/
[3]http://www.eclipse.org/MoDisco/

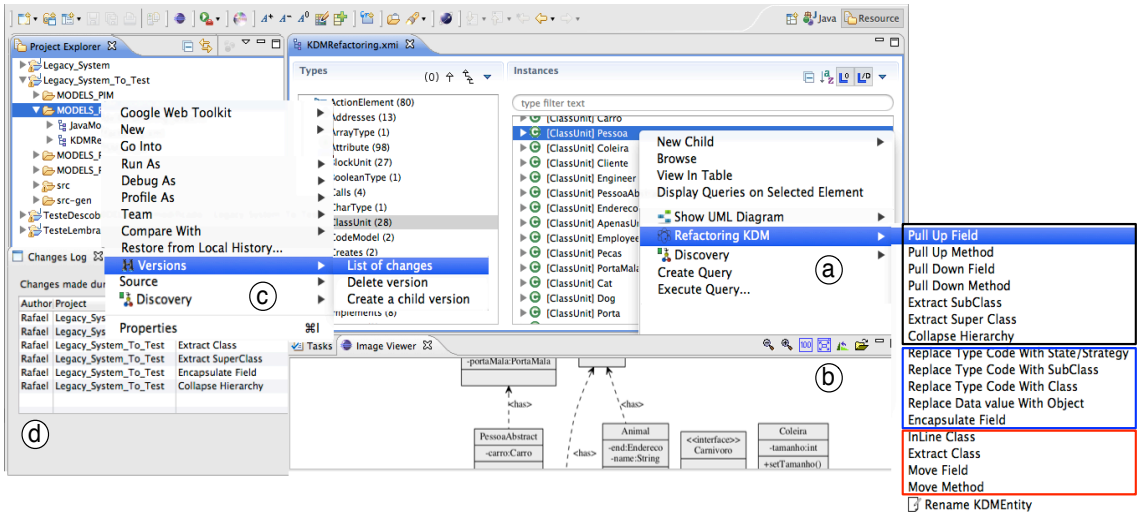

Figure 3. KDM-RE's Interface

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 ⓑ (see Figure 3) shows a class diagram that is 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 this 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 a 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 3), 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.

2.2. Executing Refactoring KDM-RE

After the engineer clicks on the menu-item (see Figure 3 ⓐ) and choose which refactoring to apply, a method run() in the class related to the chosen refactoring is being called. In this method the refactoring classes and a "RefactoringWizard" are started. In every refactoring the KDM file must be analyzed to find the meta-classes that are affected by a specific refactoring. Because of the structure of the KDM file, the easiest way to do this, is a traversal using the visitor pattern [**?**]. Therefore we implemented the visitor pattern

in KDM-RE to assist the travel of all meta-classes in a correct order.

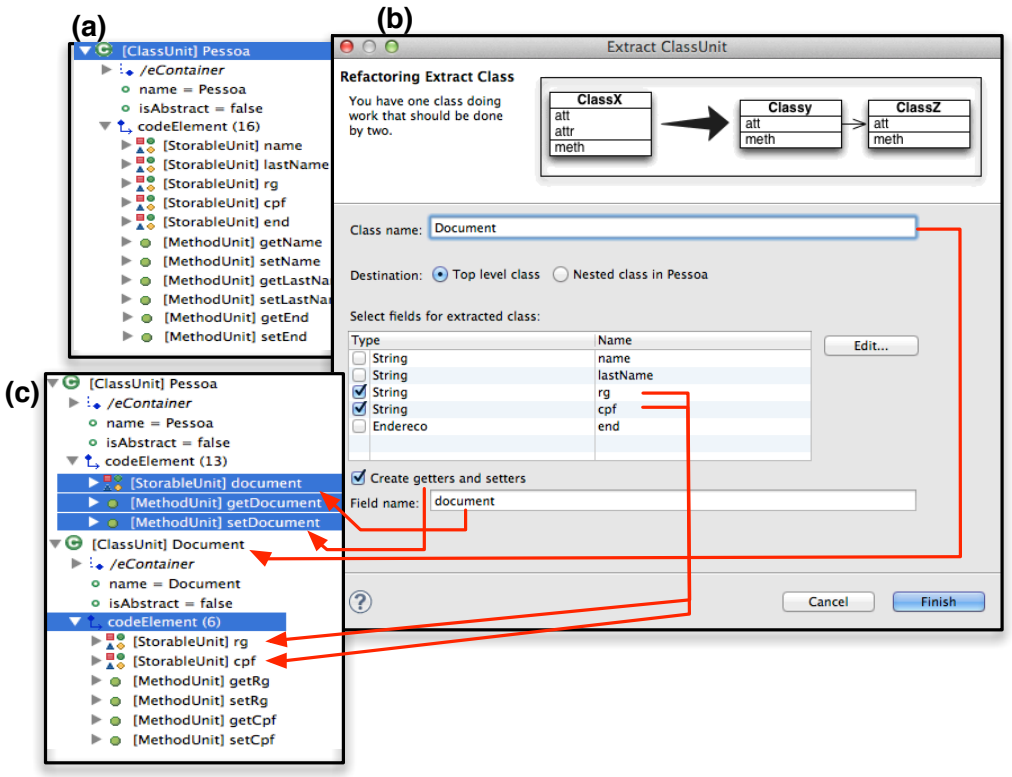

Figure 4. Extract Class Wizard

For explanation purpose pretend that the engineer found out that one class is doing work that should be done by two, thus, he must apply the refactoring "Extract Class". As stated earlier the first step is to select the metaclass that should be extracted into a separate one, this step is illustrated in Figure 4(a). After selecting the metaclass, a right-click opens the context menu where the refactoring is accessible. After the click, the system displays the "RefactoringWizard" to the engineer, Figure 4(b) depicts the Extract Class Wizard. In this wizard, the name of the new metaclass can be set. Also a preview of all detected "StorableUnits" and "MethodUnits" can be chosen to put into the new metaclass. Further, the engineer can select if either the new metaclass will be a top level metaclass or a nested metaclass. The engineer also can select if the KDM-RE must create instances of "MethodUnits" represent accessors methods (gets and sets). Finally, the engineer can set the name of the "StorableUnit" that represent the link between the two metaclasses (the old metaclass 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 4(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 4(c).

After the engineer realize the refactoring a UML class diagram is created on the fly to mirror graphically all changes performed in the KDM model, see Figure 3ⓑ. 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 3ⓓ.

2.3. Forward Engineering

After the engineer realize the refactoring the next steps are to transform the KDM model to a PSM (Java Model) and to generate the refactored source-code conforming the PSM. The former step is carried out based on a set of transformations using ATL, due space limitation these transformations are not depicted. The latter was performed by using a textual template approach, such as the Acceleo. A template can be thought of as the target text with holes for variable parts. The holes contain metacode which is run at template instantiation time to compute the variable parts.

to generate Java code from class models conforming to the meta- model in Figure 2(a).

3. Architecture

In Figure 5 is depicted the architecture of KDM-RE which is split in three layers. As shown in this figure, the first layer of our *plug-in* is the Core Framework. This layer represents that we devised the *plug-in* on the top of the Eclipse Platform. In this layer it is also possible to see that we used both Java and Groovy as programming language. Moreover, this layer contains Eclipse Plugins on which our tool is based on, such as MoDisco and EMF. We used MoDisco[4] that is 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, Eclipse Modeling Framewokr (EMF)[5] 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 *plug-in* were implemented. KDM-RE 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. KDM-RE also provides a way to create multiple versions of the KDM file to allow the engineer to assess different refactorings in the same system. In order to optimizes memory usage of multiple versions for large models and enabling to work interactively on multiple models our *plug-in* persists these models in a MongoDB. We chose MongoDB since it provides a high performance. After the engineer to choose a version refactored a forward engineering is carried out and the source code of the modernized target system is generated again. Finally, the top 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.

4. Related Work

5. Concluding Remarks

In this paper is presented the KnowDIME to support the refactoring of legacy systems based on ADM, which uses the KDM standard. It follows the theory of the horseshoe

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

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

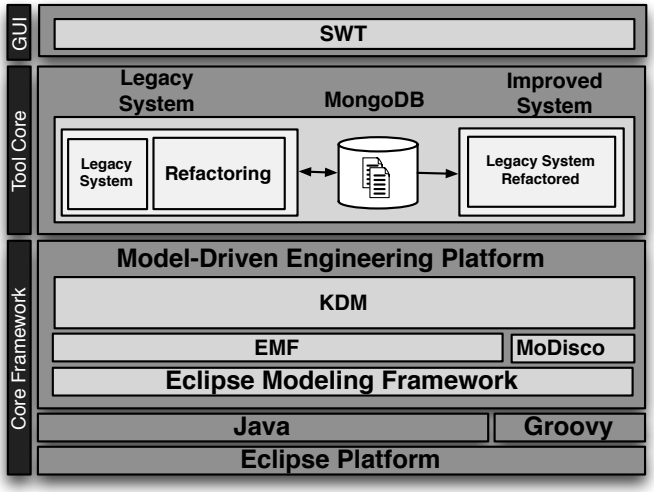

Figure 5. Architecture of KDM-RE

modernization model, which is threefold: (*i*) **Reverser Engineering**, (*ii*) **Reestructuring** and (*iii*) **Forward Engineering**.

Firstly, all the artefacts of the legacy system must be transformed into PSMs by statically analyzing the legacy source code. Still in the first step, these PSMs are integrated into a KDM model, i.e., a PIM model, through a M2M transformation implemented using ATL. Secondly, the KnowDIME applies a set of model refactorings and model optimizations in this PIM. Afterwards, KnowDIME executes a set of M2M transformation taking as input the PIM and producing as output a model conforming to the KDM models into UML meta model. Finally, an improved system is obtained from this UML by means of a set of M2C transformation; additionally, the generated code can be complemented by the software engineer in accordance with the more detailed specifications of the application business rules, such as the implementation of specific behaviors and features not covered by the code generation.

We believe that KnowDIME makes a contribution to the challenges of Software Engineering which focuses on mechanisms to support the automation of software refactoring process. Long term future work involves conducting experiments to evaluate the level of maintenance provided by KnowDIME. It is worth highlighting that KnowDIME is open source and it can be downloaded at *www.dc.ufscar.br/∼valter/crossfire*.

6. Acknowledgements

The authors would like to thank CNPq for Processes 241028/2012-4 and FAPESP for Process 2012/05168-4.

References