

# Metamodel refactoring

## Library of primitives of transformation

Olfa Ben Hadj Alaya  
INSAT  
olfabenhadjalaya@gmail.com

Walid Charfi  
INSAT  
charfiwalid@hotmail.com

Mohamed Romdhani  
INSAT  
Mohamed.Romdhani@insat.rnu.tn

Mohamed Maddeh  
ENSI  
maddeh\_mohamed@yahoo.com

INSAT, Tunisia

**Abstract:** Most refactoring tools work at code level. It's possible to define transformation operations for refactoring at metamodel level. In this paper, we discuss the current implementation of a library of primitives working on class diagrams written in standard interchange format. We also expose this library in a case study focused on applying a design pattern to a model.

*Key words: primitives, refactoring, model, interchange, transformation.*

### I. INTRODUCTION

The Model Driven Architecture (MDA), define a new approach for software designing. It was launched by The Object Management Group (OMG) in 2001.

The MDA concentrates on models instead of code. The specifications of the MDA separate in the manner of designing software between two axes. The first one defines a kind of logic which is totally independent from the implementation and the second axe is concentrated on the implementation and the used platform.

So, the MDA approach allows the realization of the same model on several platforms thanks to standardized projections. She allows also the applications to interoperate

by connecting their models and bears the evolution of platforms and techniques. [2]

The implementation of MDA approach is entirely based on models and theirs alteration.

In this paper, we focus on the conversion of models and their refactoring. Because refactoring is usually used on the code level, the MDA approach found that it could be useful to study the refactoring on the model level.

Models according to the OMG standards are represented by a format of interchange: XMI (XML MetaData Interchange). This format is the most of the time used as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels).

The particular problem we deal with is the transformation of models from a source model to a destination model without changing the observable behavior. [3]

Many studies were made in this context. Each of these

studies had their advantages and disadvantages. These ideas are proposed in the states of the art.

The rest of this paper is structured as follows. Section 3 discusses in detail our approach to the problem of model transformation. Section 4 presents an evaluation of our prototype through the application of a specific pattern. And finally, section 5 provides our conclusion.

## II. STATES OF THE ART

The refactoring work for improving the quality of a software (by eg the modularity, the legibility, the reuse, the complexity, the maintainability). The greater part of the researches made on the refactoring concentrate at the level of the code and less were interested in the phase of conception and this kind of refactoring still stays in a stadium of rather young progress.

In E. Biermann et al., 2006, Enrico Biermann et al. suggest to use the eclipse modeling framework (EMF), *a modeling framework and code generation facility for building tools and other applications based on a structured data model.* [1]

They introduce the refactoring of models based on EMF by defining rules of transformation applied to the models of EMF.

The authors of this model offer for developers the means to help them choose the most appropriate kind of refactoring for the considered model.

This approach is similar to the MDA model because it considers an EMF metamodel and uses rules of transformation to change a model into another model. Nevertheless, this solution is not generic because it is based on EMF models and not MOF models which represent the standard that provides reuse and portability.

In Rui et al., 2003, the authors applied the primitives of refactoring on models, but not all the models, just use-case models. They propose a generic refactoring but we can't consider this as a refactoring of models since it is specific to use case models. [4]

In Raul et al., 2005, the authors suggest a language for refactoring which is totally independent from the platform and using a framework based on Minimal Object-Oriented Notation, named MOON. *A refactoring engine, based on the MOON core and extensions, is responsible of checking and executing the refactoring elements on the code. Finally the refactored code is generated.*[5] In order to provide more powerful reuse capabilities, refactoring operations are defined by composition. A refactoring is composed of preconditions, actions and post conditions. These conditions are useful to check the preservation of the behavior of models. These actions, pre and post conditions are stored in a repository in order to be reused for new refactoring operations.

Various other formalisms have been proposed to understand and explore model refactoring. Most of these approaches suggest expressing model refactoring in a declarative way. Van Der Straeten et al. (2004) use description logics; Van Der Straeten & D'Hondt (2006) use a forward-chaining logic reasoning engine to support composite model refactorings. Gheyi et al. (2005) specify model refactorings using Alloy, a formal object-oriented modeling language. They use its formal verification system to specify and prove the soundness of the transformations.

## III. REFACTORING LIBRARY

The refactoring library runs primitives of transformation in order to obtain a new model from an old one already created by the Poseidon tool.

The library uses an XMI (XML Metatdata Interchange) file as an input and produces a transformed XMI file corresponding to the same metamodel. Each transformation respects established pre conditions and post conditions in order to keep the correctness of model. Here is how things concretely work:

- The library analyses and parses the class diagram contained into the input XMI file.
- User runs one to many primitives on the current class diagram.

- Lastly, user save the transformed model to an XMI file and has the possibility to work on it in the future.

Transformations can be combined into a box to enhance a particular given model, allowing, by the way, automatic refactoring process and offering ease of use for inexperienced and skilled users.

The following figure shows the transformation process.

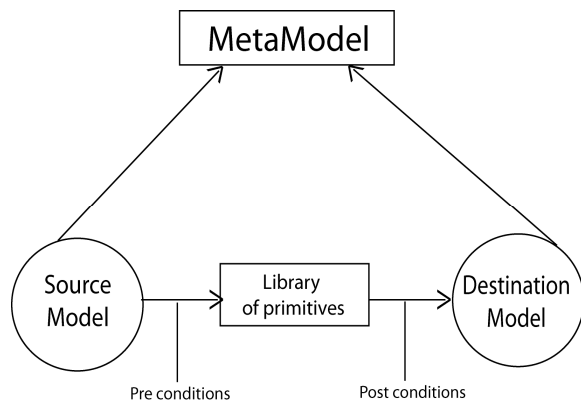


FIG. 1 Global schema of the library

### A. Refactoring repository

An XMI file is the representation of a model including the different components. So each component is represented by a specific element in the XMI file. The idea is to consider these elements and create a class for each one in order to use them freely. For example, a class for model, a class for package, a class for operation...

The following figure represents an overview of the class diagram of our application.

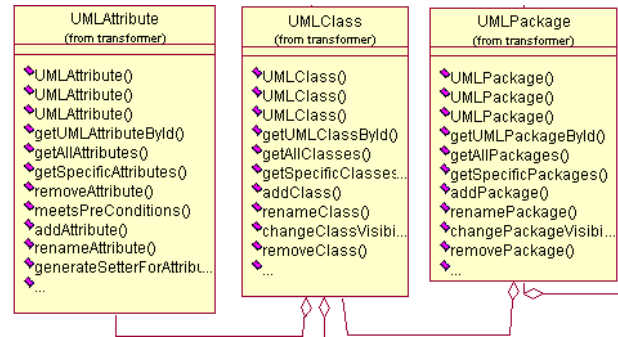


FIG. 2: Overview of repository

Refactoring primitives are implemented as methods in java classes. These methods verify the pre and post conditions respectively before and after transforming a model in order to preserve its original behavior. For example, the action **addOperation** has to verify the precondition consisting in another operation with the same signature in order to respect the polymorphism.

### B. Current State

The current version of our tool offers twenty five refactoring operations:

- Add, change type and remove parameters.
- Add, rename and remove packages, classes, methods and attributes.
- Edit the visibility of packages, classes, methods and attributes.
- Add and remove inheritance between two classes.
- Add, rename and remove association.
- Edit the multiplicity of the associations.

Our tool offers possibilities to open an XMI file and load the class diagram into a tree view. Depending on the selected node, a contextual menu can be used to apply the adequate

primitives. Lastly, the transformed diagram could be saved to an XMI file.

The following figure demonstrates how to add an association to an existent model, by specifying the association name and the first and second end multiplicities.

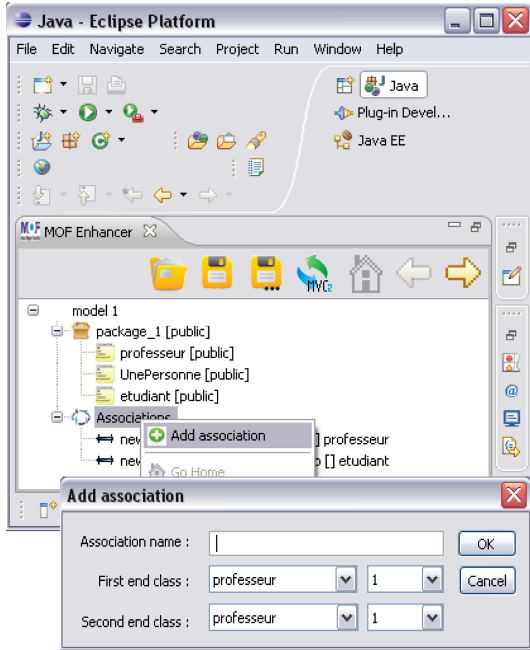


FIG. 3: Refactoring Tool

#### IV. EVALUATION

##### A. Case study

In order to put in action our contribution, we have chosen a case study focused on MVC2 (Model-View-Controller), a design pattern that separates data (model), presentation (view) and processing. The case study is about applying MVC2 to an anti-pattern (called abusively MVC0) which lacks controller and combines, into a single component, both model and view.

As a practical anti-pattern, we consider the component shown in Figure 4. This component aims at inserting information about a person (name and age) to a database and displays the result of this action.

It contains the following elements:

- Attributes describing a person,
- Insertion operation,
- And display operation.

Lastly, this component doesn't contain any controller.

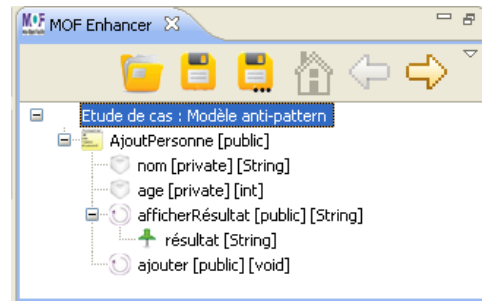


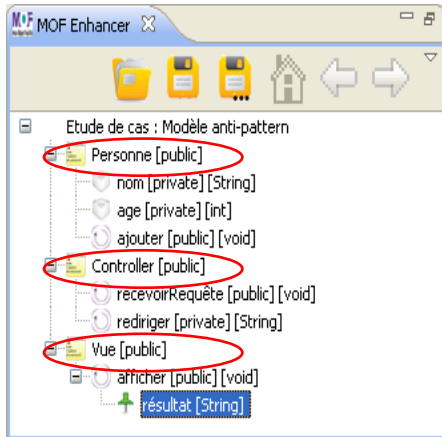
FIG.4 anti pattern to be transformed

##### B. Results

In order to enhance this anti-pattern, we create three components:

- A controller that receives request and forwards to view after processing is finished,
- A model that represents a person and contains access methods (the insertion operation in our case),
- And a view that displays the actual result of the insertion.

The figure 5 shows the result of the refactoring using the library of primitives. To obtain this new pattern we executed many primitives from the library such as **AddClass** to create the controller and also **AddOperation** to create additional methods.



**FIG. 5: Result of refactoring from MVC0 to MVC2**

The library offers the primitives that could rich the desired target.

However, some patterns, like MVC2 in our case, need a minimum of experience to know the right primitives to use and in the right order.

In this context, and for the considered pattern, we created, for this particular case study, a functionality that enables the automatic transformation from MVC0 to MVC2, making the enhancement of the model simpler and faster.

## V. CONCLUSION

The library we developed and exposed by a plug-in for Eclipse, offers approximately the most common primitives that could be applied to a class diagram. These primitives are concern refactoring of models, including actions on packages, classes, attributes, operations and parameters.

As a proof of flexibility and extensibility, other primitives and enhancement functionalities (based on existent primitives) could be added to this library, making it able to repair/enhance models.

## REFERENCES

- [1] Eclipse, EMF, <http://www.eclipse.org/modeling/emf/>, May, 2008.
- [2] Projet ACCORD, « La démarche MDA », Livrable 1.1-5, May 2002
- [3] Refactoring,SourceMaking  
<http://sourcemaking.com/refactoring/defining-refactoring>, May,2008.
- [4] Raul Marticorena, « Analysis and Definition of a Language Independent Refactoring Catalog », *17<sup>th</sup> Conference on Advanced Information Systems Engineering (CAiSE 05). Portugal.*, page 8, jun 2005.
- [5] Raul Marticorena, Carlos Lopez, and Yania Crespo, « Extending a Taxonomy of Bad Code Smells with Metrics », *WOOR'06*, Nantes, 4<sup>th</sup> July, 2006.