Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study

Thorsten Arendt, Florian Mantz, Lars Schneider, Gabriele Taentzer

Philipps-Universität Marburg, Germany, FB 12 - Mathematics and Computer Science, Software-Engineering
{arendt, mantz, taentzer}@mathematik.uni-marburg.de
Capgemini sd&m, Offenbach, Germany derlarsschneider@gmail.com

Abstract. Since model-driven development (MDD) has evolved to a promising trend in software development, models become the primary artifacts of the software development process. To ensure high model quality, using appropriate quality assurance techniques like model refactoring is an essential task. So far, tool support for model refactoring is limited, particularly for models using the Eclipse Modeling Framework (EMF). In this paper we present the results of a case study that examines three solutions for EMF model refactoring, namely the Language Toolkit (LTK), the Epsilon Wizard Language (EWL) and EMF Refactor, a new approach in the field of model refactoring.

1 Introduction

Model-driven development (MDD) has become a promising trend in software development. Here, models are in the focus of work and represent the primary artifacts in the software development process. Considering code generation, software quality depends directly on the quality of the input models. Furthermore, the Unified Modeling Language (UML) evolved to a quasi-standard, so modelers have to respect the development of high quality models, in particular UML2 models.

To obtain high quality models the existing model quality has to be determined regarding selective quality aspects of interest. During and after using appropriate quality assurance techniques an ongoing revision of this quality is required. An obvious approach for quality assurance of UML2 models is to lift software assurance techniques to the level of models where possible. Here, well-known techniques like software metrics, code smells, and code refactorings [10] have been taken into account.

A variety of tools for quality assurance of code exist, in particular for the refactoring of Java code. But for model refactoring tool support is limited so far. Since the Eclipse Modeling Framework (EMF) [1] has become a key reference in the field of MDD, it is obvious to adapt tools supporting quality assurance techniques for EMF models.

In this paper, we present three approaches for specifying and applying EMF model refactorings. We specify a sample UML2 model refactoring by means of
the Language Toolkit (LTK) [6] and the Epsilon Wizard Language (EWL) [4],
two existing solutions to handle refactorings in Eclipse. Furthermore, the case
study investigates EMF Refactor [2] which relies on EMF Tiger [3], a model
transformation tool based on graph transformation concepts. The approaches
are analyzed with respect to seven defined evaluation criteria. As a result, the
different ways of specifying and executing the sample model refactoring are dis-
cussed and the benefits and drawbacks of each approach are pointed out. In
[15], EMF Tiger and a number of other graph transformation-based tools were
compared with each other using a compact practical model transformation case
study.

The following sections are organized as follows: In Section 2, the evaluation
criteria for the case study and the sample model refactoring Change Attribute to
Association End are defined. Section 3 shows brief explanations how the sample
refactoring is specified using the investigated approaches. Their differences as
well as their benefits and drawbacks are discussed in Chapter 4. In Chapter 5
we summarize our contributions and discuss future work.

2 Case Study Description and Evaluation Criteria

This chapter introduces the sample UML2 model refactoring Change Attribute
to Association End and specifies the evaluation criteria for the case study.

2.1 Sample Refactoring

Since UML2 class diagrams are very closely related to source code, many exist-
ing code refactorings can be directly adopted to UML2 class diagrams. However,
there are few model refactorings which are specific to the model level and there-
fore cannot be adopted from code refactorings. The sample refactoring used in
this case study is one of the latter category which changes an attribute to an
association end.

Fig. 1. Sample Class Diagram before Refactoring (excerpt)

Fig. 1 shows an excerpt of a class diagram. At a first glance, one might
suppose that class Address is isolated from all other model elements. But if we
take a closer look to the model, we identify attribute address in class Customer
which is of type Address.

For a better understanding of class structures, it would be worthwhile to rep-
resent this relationship more explicitly. This can be achieved by applying model
refactoring Change Attribute to Association End. After refactoring application,
attribute address of class Customer will be depicted as an association end (see
specification of UML2.1 [5]).
2.2 Evaluation Criteria

Each approach is investigated considering the following evaluation criteria. For each criterion questions are defined which are evaluated during the specification and execution of sample refactoring solutions.

Refactoring Specification

- **Complexity** - How complex is the effort to specify the refactoring? Are there ways to reduce this effort? Here, LoC and the number of specified rules have to be compared.
- **Correctness** - Is it possible to specify a refactoring which application results in an inconsistent model? Are there any precautions to avoid it?
- **Testability** - Which effort is needed to test the specified refactoring in detail? Are there ways to automate these tests?
- **Modularity** - Can the specified refactoring be combined with other refactorings? (This is an important aspect when defining more complex refactorings by reusing existing ones.)

Refactoring Application

- **Interaction** - How convenient is it to apply the refactoring? Are there any facilities to simplify user inputs? Here, differences considering UI features have to be evaluated from a (subjective) user’s point of view.
- **Features** - Does the refactoring provide a preview in order to be able to cancel or commit the refactoring? Does it provide undo and redo functionality?
- **Malfunction** - What happens if the appropriate refactoring cannot be executed in the given situation? Are there reasonable error messages?

3 Case Study Execution

The sample model refactoring was implemented using LTK, EWL and EMF Refactor. Due to space limitations the implementations are presented in a very compact form. The entire specifications (source code, rules, etc.) can be found on the EMF Refactor web site [2].

3.1 Refactoring Implementation using LTK

The Language Toolkit (LTK) [6] is a language neutral API to specify and execute refactorings in an Eclipse-based IDE. So it is possible to handle EMF model refactorings by LTK. The API can be found in the org.eclipse.ltk.core.refactoring and org.eclipse.ltk.ui.refactoring plug-ins. The API classes of LTK incorporate an exact, predefined procedure for refactorings in Eclipse.
For specifying the sample model refactoring 7 classes have to be implemented. During implementation it became obvious that only 4 classes are refactoring specific (RefactoringInfo, RefactoringInputWizardPage, RefactoringAction, and RefactoringProcessor). Classes EMFChange and Refactoring are generic for EMF model refactorings. Class RefactoringWizard is refactoring specific only since it initializes RefactoringInputWizardPage. The refactoring specific classes are:

- **RefactoringInfo** - This class manages all required informations like the selected Property object, the name of the new association and the name of the association’s ownedEnd property.

- **RefactoringInputWizardPage** - This class is responsible for displaying and handling the required user input (name of the new association and name of the association’s ownedEnd property).

```java
RefactoringStatus result = new RefactoringStatus();
Property property = this.refInfo.getProperty();
if (property.getType() != null) {
    if (property.getType().isInstance(Class.class)) {
        if ((this.refInfo.getProperty()).getAssociation() != null)
            result.addFatalError("The selected Property is already an association end!");
        else
            result.addFatalError("The selected Property does not have a type!");
    } else
        result.addFatalError("The selected Property is not a Class!");
}
return result;
```

**Fig. 2.** LTK: method body `RefactoringProcessor.checkInitialConditions()`

- **RefactoringAction** - This class is responsible for refactoring initiation. It sets the selected Property and initializes instances of RefactoringWizard, RefactoringProcessor, Refactoring, and RefactoringInfo. The refactoring is initiated by invoking method `RefactoringWizardOpenOperation`::run(). The extension point `org.eclipse.ui.popupMenus` is served by this class.

```java
Map.Entry<Object, List<FeatureChange>> entryAsName = createEObjectToChangesMapEntry(ac);
FeatureChange fCasName = createFeatureChange();
fCasName.setNameFeatureName("name");
fCasName.setValue(this.refInfo.getAssociationName());
entryAsName.setValue().add(fCasName);
changeDescription.getEObjectChangeSet().add(entryAsName);
```

**Fig. 3.** LTK: method `createChange()` (excerpt)

- **RefactoringProcessor** - This is the main class for executing the sample refactoring. Method checkInitialConditions() checks whether the type of the selected Property is an instance of Class and whether it is not already part of an Association (see Fig. 2). The most important method of class `RefactoringProcessor` is `createChange()`. This method creates an instance of EMFChange by generating a `ChangeDescription` that describes

---

3. `org.eclipse.ltk.ui.refactoring.RefactoringWizardOpenOperation`

4. `org.eclipse.emf.ecore.change.ChangeDescription`
all required model changes and is also used for undo and redo functionality.

Fig. 3 shows an excerpt of method `createChange()`. Here, feature `name` of the newly created `Association` is set to the appropriate String managed by the `RefactoringInfo` object.

3.2 Refactoring Implementation using EWL

The Epsilon Wizard Language (EWL) is an integral part of Epsilon [13], a platform for building consistent and interoperable task-specific languages for model management tasks. For this purpose, Epsilon consolidates common facilities in a base language, the Epsilon Object Language (EOL) [11], that new task-specific languages can reuse.

EWL is a tool-supported language for specifying and executing automated model refactorings which the authors of EWL call update transformations in the small [12]. These model refactorings are applied on model elements that have been explicitly selected by the user. Epsilon provides an Eclipse-based interpreter that can execute programs written in EWL.

In EWL, the sample refactoring has been implemented as follows: First, the type of the selected model element has to be checked to be a `Property` of type `Class`. Furthermore, this property does not already have to be part of an `Association`. These preconditions are checked in the `guard` section of the EWL program. Variable `self` refers to the model object which is used to invoke the refactoring and is a `Property` in this example. If the guard conditions fail, the refactoring will not be performed. Fig. 4 shows the `guard` section of the EWL solution.

The next step is to specify the label that will be provided to the user in the context menu of the selected model element. This is done in the `title` section of the EWL program.

The final and most important part of the EWL solution is the `do` section that specifies the effects of the refactoring when applied to a compatible selection of model elements (see Fig. 5). Regarding the sample refactoring we have to organize the required user input first, in particular the name of the new association and the name of the association’s `ownedEnd` property.

After obtaining the user input all necessary new objects are created and the appropriate features are set. These are in particular:

---

```
3  guard {
4    if (self isKindOf(Property)) {
5      if (self.type isDefined) {
6        if (self.type isKindOf(Class)) {
7          return self.association.isUndefined;
8        } else { return /error; }
9      } else { return /error; }
10    } else { return /error; }
11  }
```

---

```
var upperVal : new LiteralInteger;
upperVal.value = 1;
var lowerVal : new LiteralInteger;
lowerVal.value = 1;
var ownedEndP : new Property;
ownedEndP.name = self.name;
ownedEndP.type = self.type;
ownedEndP.upperValue = upperVal;
ownedEndP.lowerValue = lowerVal;
var asso = new Association;\nasso.name = associationName;
asso.ownedEnd.addAll(ownedEndP);
self class package.packagedElement.add(asso);
```
– A new Property with features name, type, upperValue, and lowerValue (that are set to 1 each).
– A new Association with features name, ownedEnd, and memberEnd.

Again, global variable self is used to get the appropriate features of the selected Property. Finally, the new association has to be added to the including package.

3.3 Refactoring Implementation using EMF Refactor

A new approach to specify and execute EMF model refactorings is EMF Refactor [2]. The development of new refactorings in EMF Refactor is based on EMF Tiger [3] [8], an Eclipse plug-in that performs in-place EMF model transformations [7] [14]. The model transformation concepts of EMF Tiger are based on algebraic graph transformation concepts. It provides a graphical editor for the design of transformation rules and a Java code generator which has been extended by EMF Refactor.

Model refactorings are designed by ordered sets of rules. Each rule describes an if-then statement on model changes. If the pattern specified in the left-hand side (LHS) exists, it is transformed into another pattern defined in the right-hand side (RHS). Here, several input parameters can be used to specify the LHS pattern in more detail. Additionally, several negative application conditions (NACs) can be specified which represent patterns that prevent the rule from being applied. Mappings between objects in LHS and RHS and/or between objects in LHS and NACs are used to express preservation, deletion, and creation of objects. A LHS object being mapped to a RHS object is preserved, while an object without mapping to a RHS object is deleted from the model including all its possible children. A RHS object without an original LHS object is newly created and attached to the model.

Fig. 6. EMF Refactor: LHS

To specify the sample refactoring in EMF Refactor we have to define one rule. The LHS of this rule is shown in Fig. 6. This pattern represents the abstract syntax which has to be found when starting the refactoring from within the context menu of a Property named propName whose type is a Class. To ensure that the selected Property is not already part of an Association an appropriate NAC is defined, that is similar to the LHS but with an additional Association instance that references the selected Property as memberEnd (not shown here).

Fig. 7 shows the RHS of the sample refactoring rule. It contains a new Association object with a new opposite association end (Property). This end
is equipped with multiplicity 1 as lower and upper bound. The newly created objects are named by additional input variables associationName and srcProperty.

During rule specification one can show that the specified transformation rule is consistent. This means that applying the appropriate EMF model transformation always leads to EMF models consistent with typing and containment constraints. To do so, you have to check whether the rules perform restricted changes of containments only. Consistent EMF model transformations behave like algebraic graph transformations. Hence, the rich theory of algebraic graph transformation can be applied to show functional behavior and correctness [9]. The sample refactoring rule is consistent because all new object nodes (Association, Property, and two LiteralIntegers) are connected immediately to their according container (see Fig. 7).

After rule definition the corresponding refactoring code is generated, including a wizard for parameter specification. Here, default values for the parameters associationName and srcProperty are set.

4 Case Study Results and Evaluation

This chapter presents the results of the case study. First, all three solutions are compared along the criteria introduced in Chapter 2 and finally they are interpreted.

Complexity - All approaches require a comprehensive understanding of the UML2 meta model [5]. In LTK, 7 Java classes including 711 LoC were implemented. 416 LoC can be generated and 195 are refactoring specific, in particular methods checkInitialConditions() and createChange() of class RefactoringProcessor. Here, the most challenging task is to exactly implement the corresponding ChangeDescription object because of its general and complex API. In EWL, one single file with 47 LoC was implemented which is a much better value than LTKs 195 LoC. Automatically generating generic parts would not lead to a significant reduction. Finally, in EMF Refactor the whole refactoring
code was generated from one rule only containing 32 objects (EClasses and textttEReferences). This number of objects is hardly comparable to any LoC value. However, specifying refactorings using EMF Refactor seems to be similar extensive than using EWL. Additionally, in EMF Refactor individual parameter settings for code generation are supported by a convenient wizard.

Correctness - In LTK, an incorrectly specified ChangeDescription object would lead to an inconsistent model after executing the refactoring. There are no known precautions available to avoid this. Since all model changes in EWL are directly implemented, there is also no special support to specify refactorings which yield consistent models only. EMF Refactor instead uses EMF Tiger that provides consistency checks regarding containment and multiplicity issues. This is done using the underpinning graph transformation concepts. Hence, it is almost impossible to specify transformations, especially refactorings leading to inconsistent models.

Testability - A specified refactoring has to be tested by applying it to various models that represent possible situations. Since every refactoring in LTK is a single Eclipse plug-in, it is very time-consuming to start a new Eclipse instance after each code change. This tasks could be significantly facilitated by generating test code or using PDEUnit, which is a test framework for Eclipse plug-ins. Because EWL is an interpreted language, testing is not that time-consuming and a straightforward task. Nevertheless, there is no known way to automate this. For EMF Refactor the same comments as for LTK hold. Here, a first approach for generating tests using JUnit is available.

Modularity - Since all model changes in LTK are directly implemented in Java, it seems to be possible to combine several existing refactorings to more complex ones by passing required parameters, and adapting conditions, and Change Descriptions. Here, it is necessary to develop an advanced approach to support this features. In EWL, there is no known way to combine refactorings so far, except for copying and adapting code of existing ones. Although EWL supports operations that can be reused, it does not support reusability of wizards. For EMF Refactor the same comments as for LTK hold, but it seems to be less promising than combining refactorings in LTK. Again, a first approach to combine so called basic refactorings to more complex ones is under development, but it is still not available.

Interaction - All approaches provide the selection of refactorings via the context menu of a Property element in the standard EMF instance editor. EWL additionally supports UML2Tools which can be supported by the others as well if a further extension point is served. The refactoring wizard page of LTK provides one input line for each required parameter. Each parameter has a specified default value. In EWL, the context menu has an entry specific to the name of the selected Property. All parameters are entered in separate dialogs including specified default values. For EMF Refactor the same comments as for LTK hold.

Features - In LTK, after parameter editing the wizard provides an optional preview of the model changes made by the refactoring. The preview is provided by EMF Compare. Undo/Redo functionality is supported. In EWL, there is no
preview available, but Undo/Redo functionality is supported. After parameter editing in *EMF Refactor* the wizard always shows a preview of possible model changes when executing the refactoring. Again, this is provided by EMF Compare. Undo/Redo functionality is not supported.

**Malfunction** - If a certain precondition in *LTK* fails, a message box including a reasonable error message is shown as specified in method `checkInitialConditions()` of class `RefactoringProcessor`. *EWL* provides the refactoring only, if all preconditions specified in the `guard` section hold. After parameter input in *EMF Refactor*, the user is informed when the refactoring can not be executed because of violated conditions. This is merely done by the generic message *The refactoring changed nothing at all*. Each solution requires non-empty parameters, more precisely names for the new model elements *Association* and *Property*.

<table>
<thead>
<tr>
<th>Goal</th>
<th>LTK</th>
<th>EWL</th>
<th>EMF Refactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>o</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Correctness</td>
<td>o</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Testability</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>Modularity</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Interaction</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Features</td>
<td>+</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>Malfunction</td>
<td>+</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>

Table 1. Results of the Case Study

Table 1 summarizes the results of the case study. Each approach has been evaluated and marked as follows:

- The approach meets the evaluation criterion: +
- The approach does not meet the evaluation criterion but is still moderate: o
- The approach does not meet the evaluation criterion at all: -

Each approach has its individual strengths and weaknesses. *LTK* provides permanent positive results when executing the model refactoring. This is not astonishing because *LTK* was developed to unify refactoring processes in Eclipse. However, *EMF Refactor* and *EWL* are more suitable for specifying EMF model refactorings. *EMF Refactor* impresses by its graphical nature of defining model transformations and its underlying graph transformation concepts. Otherwise, *EWL* seems to be the best choice when specifying refactorings in a textual way.

5 Summary and Future Work

In this paper we present the results of a small case study that examines three options for EMF model refactoring, namely the Language Toolkit (LTK), the Epsilon Wizard Language (EWL) and EMF Refactor, a new approach in the field of EMF model refactoring. The study demonstrates that each approach has its individual strengths and weaknesses. *LTK* is the leading approach during model refactoring application, whereas *EMF Refactor* and *EWL* are promising approaches in specifying EMF model refactorings.
As a conclusion of the presented case study, it looks worthwhile to check whether the approaches can be combined in a way that merges their benefits. Such a combination (EWL/LTK or EMF Refactor/LTK) seems to be a promising way to go. Here, first steps are done by restructuring the architecture of EMF Refactor. EMF model refactorings will be executed using LTK, whereas the specification will be done by other approaches (for example the previous version of EMF Refactor, EWL, or a completely different one).

In the presented case study we inspected specification and implementation of one single UML model refactoring. Further studies should be performed inspecting several other model refactorings which are more complex or even based on other domains than UML. Additionally, further issues like incremental generation of the implementation and change propagation of source code manipulations should be included, since all investigated approaches do not consider these aspects.

References