Integration of Smells and Refactorings within the Eclipse Modeling Framework *

Thorsten Arendt   Gabriele Taentzer

Philipps-Universität Marburg
Department of Mathematics and Computer Science
Hans-Meerwein-Strasse
D - 35032 Marburg, Germany
{arendt,taentzer}@mathematik.uni-marburg.de

Abstract

Models are primary artifacts in model-based, and especially, in model-driven software development processes. Therefore, software quality and quality assurance frequently leads back to the quality and quality assurance of the involved models. In our approach, we propose a model quality assurance process that is based on static model analysis and uses techniques like model metrics and model smells. Based on the outcome of the model analysis, appropriate model refactoring steps are performed. Appropriate tools support the included techniques, i.e. metrics, smells, and refactorings, for models that are based on the Eclipse Modeling Framework (EMF). In this paper, we present the integration of the two model quality tools *EMF Smell* and *EMF Refactor*. This integration provides modelers with a quick and easy way to erase model smells by automatically suggesting appropriate model refactorings, and to get warnings in cases where new model smells come in by applying a certain refactoring.

Keywords model quality, model smell, model refactoring, Eclipse Modeling Framework

1. Introduction

Model-based software development is becoming more and more widespread in modern software projects, since it promises an increase in the efficiency and quality of software artifacts and development processes. Especially in model-driven software development, high code quality is possible only if the quality of input models is already high. Typical quality assurance techniques for models are model metrics and refactorings, see e.g. [14, 16, 18, 21]. They originate from corresponding techniques for software code by lifting them to models. Especially class models are closely related to programmed class structures in object-oriented programming languages such as C++ and Java. For behavior models, the relation between models and code is less obvious. Furthermore, the concept of code smells [13] can also be lifted to models leading to model smells.

In [7], we present the integration of these techniques in a predefined quality assurance process that can be adapted to specific project needs. It consists of two-phases: Before a software project starts, project- and domain-specific quality checks and refactorings have to be defined. Quality checks are formulated by model smells which can be specified e.g. by model metrics and anti-patterns that are defined on the abstract syntax and matched to concrete instances thereafter. After formulating quality checks, the specified quality assurance process can be applied to concrete software models by computing model metrics, reporting all model smells and applying model refactorings to erase smells that indicate clear model defects. However, we have to take into account that also new model smells can come in by refactorings. To automate some tasks of this project-specific model quality assurance process, we implemented tools supporting the included techniques metrics, smells, and refactorings for models that are based on the Eclipse Modeling Framework (EMF)[1] [20], a common open source technology in model-based software development.

In this paper, we concentrate on the integration of model quality tools *EMF Smell* [4] and *EMF Refactor* [5]. More specifically, we present mechanisms to provide modelers with a quick and easy way (1) to erase model smells by automatically suggesting appropriate model refactorings, and (2) to get warnings in cases where new model smells occur due to applying a model refactoring.

* This work has been partially funded by Siemens Corporate Technology, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WRT June 01 2012, Rapperswil, Switzerland
Copyright © 2012 ACM 978-1-4503-1500-5…$10.00

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

The paper is structured as follows: Starting with an overview of the existing tools and an example scenario in Sections 2 and 3, we present our approach and its implementation in Section 4. Thereafter, we discuss related work in Section 5 and conclude in Sections 6.

2. EMF Smell and EMF Refactor

In this section, we briefly describe the main concepts of the already existing but so far stand-alone model quality tools *EMF Smell* and *EMF Refactor*. Furthermore, we give an overview on already implemented UML2EMF smells and refactorings.

2.1 Common architecture

EMF Smell and EMF Refactor are based on the Eclipse Modeling Framework, i.e. each tool can be used on arbitrary models whose meta models are typed over EMF Ecore, for example domain-specific languages, common languages like UML2EMF used by Eclipse Papyrus[2] and the Java EMF model used by JaMoPP[3] and MoDisco[4], or even Ecore instance models themselves.

Each model quality assurance tool consists of two components. For each specific model smell or refactoring, the *Specification Module* generates Java code that can be used by the second component, the *Application Module*, for smell detection respectively refactoring execution thereafter. The application module of each tool provides project-specific configurations to select those smells and refactorings that are suitable for the specific modeling purpose. Furthermore, smells and refactorings can be concretely specified by several approaches (see next sections).

2.2 EMF Smell

EMF Smell [4] supports the specification and detection of smells in arbitrary EMF-based models. The checking process is triggered either from within the context menu of the model file to be analyzed, or it can be started on any model element in the tree-based EMF instance editor or in a graphical editor generated by GMF/GMP[5]. For example, if the checking process is started from the context menu of a specific UML package then only those smells are reported which occur inside this package.

After starting the detection process, EMF Smell checks the existence of each configured model smell wrt. the corresponding meta model and presents the detected smells in a special result view (compare Section 3). In this view, each model smell found contains a time stamp to trace detected smells over time. Furthermore, EMF Smell provides an XML export of its analysis results.

[2] http://www.eclipse.org/modeling/mdt/papyrus/

[3] http://www.jamopp.org

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

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

We have implemented altogether 105 metrics and 30 smells for UML2EMF models, more specifically for class diagrams, state chart diagrams, and use case diagrams. These are, for example:

- *Concrete Superclass*: The model contains an abstract class with a concrete superclass [16].
- *Long Parameter List*: The model contains an operation with too many input parameters [13].
- *Equal Attributes in Sibling Classes*: Each sibling class of the owning class of an attribute contains an equal attribute (same names and types, but potentially different visibilities).
- *Unused Use Case*: The model contains a use case that is not associated to any actors [1].

EMF Smell currently supports four concrete mechanisms for model smell specification. As more common approaches, (1) pure Java code or (2) OCL expressions can be used. Some smells can be detected by (3) metric benchmarks. Here, appropriate model metrics provided by the tool *EMF Metrics* [6] can be used and the corresponding benchmark is set in the project specific configuration. Pattern-based smells (i.e., smells that are detectable by the existence of specific anti-patterns in the abstract model syntax) can be specified using (4) pattern-rules formulated in the EMF model transformation tool Henshin[6] [3]. Then, Henshin's pattern matching algorithm is used to detect rule matches that can be found in the model. The matches found represent the existence of the corresponding model smell.

2.3 EMF Refactor

EMF Refactor[7] [5] is an Eclipse incubation project in the Eclipse Modeling Project consisting of three main components. Besides a code generation module and a refactoring application module, it comes along with a suite of predefined EMF model refactorings for UML2EMF and Ecore models. Currently, 22 refactorings for Ecore and 29 refactorings for UML2EMF have been specified. Some of them are intermediate parts of larger (composite) refactorings. Example UML2EMf refactorings are:

- *Create Associated Class*: Creates an empty class and connects it with a new association to the source class from where it is extracted. The multiplicities of the new association is 1 at both ends [17].
- *Pull Up Attribute*: Removes an attribute from a class or a set of classes and inserts it to one of its resp. their superclasses [10, 17].
- *Introduce Parameter Object*: Replaces a list of parameters which naturally go together by one object that is created for that purpose [13].

[6] http://www.eclipse.org/modeling/emft/henshin/

[7] http://www.eclipse.org/modeling/emft/refactor/

- *Merge States*: Merges two interconnected states of a state diagram by moving all actions from the parameter state to the contextual state, redirecting all external transitions of the parameter state to the contextual state, removing all inner transitions in between both states, and finally deleting the parameter state [11].

The application of a model refactoring in EMF Refactor mirrors the three-fold specification of refactorings based on the Eclipse Language Toolkit (LTK)[8]. After specifying a trigger model element, refactoring-specific basic conditions are checked (LTK's *initial condition check*). Then, the user has to set all parameters and EMF Refactor checks whether the user input does not violate further conditions (LTK's *final condition check*). In case of erroneous parameters a detailed error message is shown. If the final check has passed, EMF Refactor provides a preview of the changes that will be performed by the refactoring using EMF Compare[9]. Last but not least, these changes can be committed and the refactoring can take place (LTK's *model change*).

Since EMF Refactor uses the LTK technology mentioned above, a concrete refactoring specification requires up to three parts (i.e., specifications for initial checks, final checks, and the proper model changes). EMF Refactor currently supports four concrete mechanisms for EMF model refactoring specification. As in EMF Smell, refactorings can be specified using (1) Java or (2) OCL expressions (for precondition checking). The most prominent way to specify a model refactoring is to use the (3) EMF model transformation tool Henshin. Here, EMF Refactor uses Henshin's model transformation engine for executing the refactoring as well as Henshin's pattern matching algorithm to detect violated preconditions. Finally, current work concentrates on a (4) combination of existing refactorings to more complex ones by using a dedicated domain-specific language.

3. Running example

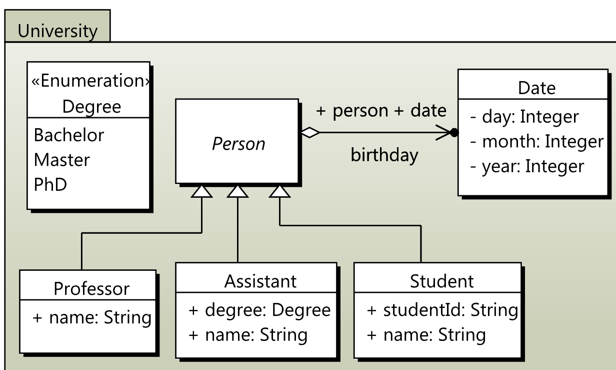

Figure 1. Example UML class diagram *University*

[8] http://www.eclipse.org/articles/Article-LTK

[9] http://www.eclipse.org/emf/compare/

In this section, we motivate our approach by the smell analysis of a small example UML class model. Figure 1 shows a class diagram used for modeling a small part of the *university* domain.

In a university there are several `Persons`, more specifically `Professors`, their `Assistants`, and `Students`. Each person has a *birthday* (of type `Date`), each student has a unique *studentId* (of type `String`), and each assistant has a *degree* which is either `Bachelor`, `Master`, or `PhD`. Furthermore, classes `Professor`, `Assistant`, and `Student` have an attribute *name* of type `String` each. This redundantly modeled attribute exactly represents an occurrence of UML model smell *Equal Attributes in Sibling Classes* (compare Section 2.2). A static quality analysis of model *University* by using EMF Smell results in a smell report as depicted in Figure 2. This report shows that smell *Equal Attributes in Sibling Classes* occurs altogether three times. The affected model elements are (as expected): class `Professor` and its attribute *name*, class `Student` and its attribute *name*, and class `Assistant` with its attribute *name*, respectively.

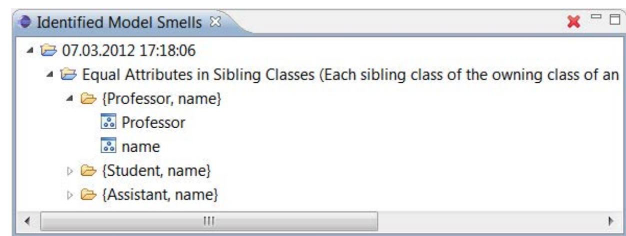

Figure 2. Smell report after analyzing UML class diagram *University*

Until now, it has been up to the modeler (respectively model reviewer) to interpret these results with respect to appropriate model changes, e.g. by applying suitable model refactorings. *So far, there are no mechanisms to provide modelers with a quick and easy way to erase model smells by automatically suggesting appropriate model refactorings. Furthermore, the potential risk of inserting new model smells when applying a certain model refactoring has not been addressed by the existing tools so far.*

4. From smells to refactorings and back again

In this section, we present our approach to the integration of model smells and model refactorings. We discuss three alternative solutions (manual integration, interrelation analysis at application time, and interrelation analysis at specification time) and present the implementation of the first two approaches by using the example model shown in Figure 1.

4.1 Manual integration

In order to propose suitable refactorings respectively to inform about potential new smells, the integration tool must be provided with information on the relations between model

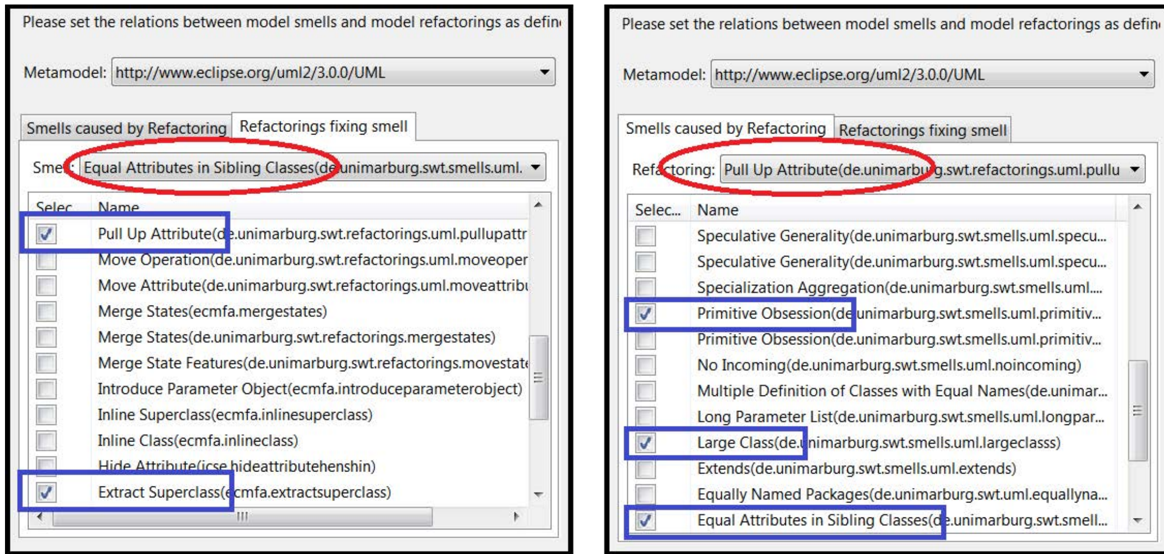

Figure 3. Manual configuration of relationships: suitable refactorings to erase a specific smell (left); possible new smells after applying a specific refactoring (right)

smells and model refactorings. The most pragmatic way is to manually define them. Here, the advantage is that the designers can adjust the implementation of model smells and model refactorings to the fact that they are going to be related. In contrast to an automated extraction of inter-relations, they can pay attention to the semantics in addition.

Since there are two possible relationships for model smells and model refactorings there have to be two relations to represent them. Our implementation uses the extension point technology of Eclipse and provides two extension points for the manual definition of relations between model smells and model refactorings. Since EMF Smell and EMF Refactor identify smells respectively refactorings by distinct identifiers, these extension points require relations from *smell IDs* to a list of *refactoring IDs* (in case of providing suitable refactorings for a given smell) and relations from *refactoring IDs* to a list of *smell IDs* (in case of possible new smells when applying a given refactoring).

The definition of relations between model smells and model refactorings can be done in two ways, directly (by serving the corresponding extension point) or via a dedicated property page of a certain Eclipse plugin project in the workspace. This property page provides graphical user interfaces for (de-)activating appropriate relations. Figure 3 shows the two interfaces for defining appropriate relations between existing model smells and model refactorings. On the left-hand side of Figure 3 an example selection of suitable refactorings for a given smell is shown. Here, we address smell *Equal Attributes in Sibling Classes* for UML models (see meta model selection on the top). For this smell two refactorings are selected to be suitable for erasing it. Refactoring *Pull Up Attribute* [10, 17] can be used to move this attribute to an existing parent class and to remove all equal attributes from the sibling classes. If the attribute should not be moved to an existing parent class, refactoring *Extract Superclass* [13, 17] can be used to create a new parent class and to pull up the attribute to it. The right-hand side of Figure 3 shows an example for the selection of potentially new smells after applying a specific refactoring. Here, the afore mentioned UML refactoring *Pull Up Attribute* is addressed and three smells are specified which can occur after applying the refactoring:

- Smell *Primitive Obsession* [13] can occur when the moved attribute and other existing attributes of the parent class primarily use primitive data types like `String` or `Integer` to encode data that would be better modeled as a separate class.
- Smell *Large Class* [8, 13] can occur when the parent class owns too many features (attributes and operations) after the refactoring with respect to a specified threshold.
- The already discussed smell *Equal Attributes in Sibling Classes* can occur but then in another context (the parent class).

In a proof-of-concept integration we have related altogether 16 UML2EMF smells to 18 potentially suitable refactorings and 14 refactorings to 6 potentially occurring smells.

4.2 Interrelation analysis at application time

The goal of an analysis at application time is to calculate possibly applicable refactorings without having to define relations manually. It is also aiming at finding caused model smells after knowing all the input for a model refactoring.

For finding applicable refactorings to a given group of model elements, this approach uses the initial check of EMF Refactor as explained in Section 2.3. Technically, the occur-

rence of a model smell detected by EMF Smell is a list of instances of class `EObject` (from EMF's Ecore meta-model) and the initial check performed by EMF Refactor exactly requires instances of class `EObject` as input (i.e. the contextual elements). So, finding applicable refactorings for a given occurrence of a model smell can be done by performing the initial check of each model refactoring that is defined for the model's meta model and for a specified context. If the initial check is successful, the model structure allows the model refactoring to be applied. However, this does not take into account the further parameters that may have to be provided by the user. Those can still make the refactoring application fail (final check, see Section 2.3). Each applicable refactoring and the model elements to which they can be applied, are stored in a map and used as the input for the suggestion dialog providing the user with a list of applicable refactorings (see an example in next section).

To address the opposite relationship between model smells and refactorings, i.e. to provide some brief information on possibly newly inserted smells after refactoring application, the application module of EMF Refactor is extended by a quantitative analysis of changes of model smell occurrences. Similar to the model preview provided by EMF Compare, this analysis provides the modeler with the total number of occurrences of model smells before and after a potential application of a given model refactoring (see an example in the next section). It thereby helps with the decision whether or not a refactoring application would improve the overall model quality or would it even make worse.

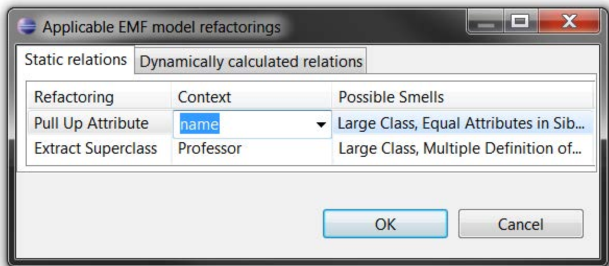

Figure 4. Refactoring suggestion dialog: manual relations

4.3 Example application

Starting from the results view of EMF Smell (see Figure 2) our proof-of-concept implementation provides a suggestion of possible refactorings according to the pre-defined relations and a dynamic analysis of applicable refactorings.

The suggestion dialog is started from within the context menu of a corresponding smell occurrence (for example, occurrence $\{Professor, name\}$, compare Figure 2) by selecting *Suggest refactorings* and consists of two tabs. The first tab (see Figure 4) suggests all model refactorings that (1) have been manually defined as being suitable to erase the corresponding model smell and (2) are also applicable in the given context, i.e. the initial check has passed. In our case, refactoring *Pull Up Attribute* could be applied on attribute *name*, whereas refactoring *Extract Superclass* could be applied on class *Professor*. Furthermore, the dialog informs about possible new smells which may be inserted when applying the refactoring (according to the configuration in Figure 3).

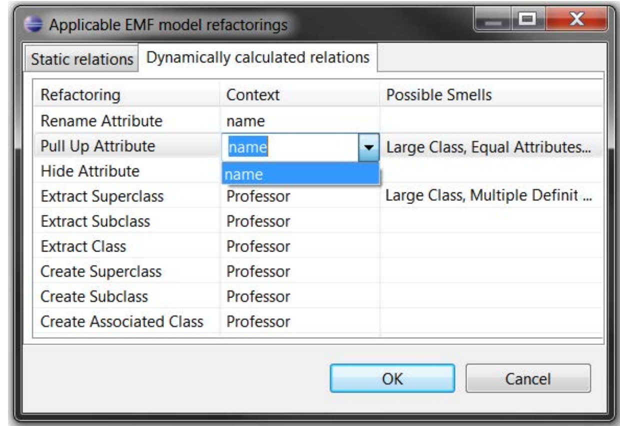

Figure 5. Refactoring suggestion dialog: analysis at application time

The second tab lists all those model refactorings which have been proven to be applicable on at least one model element in the selected smell occurrence (see Figure 5). As mentioned in the last section, this does not necessarily mean that each refactoring will potentially improve the model quality by erasing a model smell. It simply means that the target model structure allows the application of the refactoring. Again, information about possible new smells which may occur when applying the refactoring are presented according to the manual configuration.

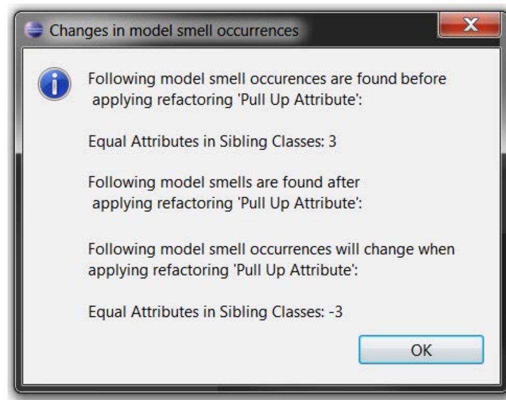

Figure 6. Information dialog: smell analysis at application time

The application of a selected model refactoring can be started directly from the suggestion dialog and the affected model element is passed as context parameter. Then, the refactoring is executed by EMF Refactor (see Section 2.3) except for the opportunity to get a quantitative analysis on changes of model smell occurrences as described in the previous section. Figure 6 shows the corresponding information

dialog when applying refactoring *Pull Up Attribute* on attribute *name* in our example model. Before the refactoring, smell *Equal Attributes in Sibling Classes* occurs three times; after refactoring these occurrences will be eliminated and no further smell will be inserted.

4.4 Interrelation analysis at specification time

The interrelation analysis at application time answers the question of applicability of model refactorings on a given model smell occurrence (see Section 4.2). This, however, always depends on the concrete model instance that is currently under inspection. The same is true for the quantitative analysis which focuses on the impact of model refactoring applications on the quality of a model (see also Section 4.2). To answer these questions independently of a given model instance is possible only by inspecting the concrete specifications of both, a model smell and a model refactoring. Whereas EMF Smell and EMF Refactor support several specification mechanisms for model smells respectively model refactorings (see Sections 2.2 and 2.3), our first reflections on an approach for interrelation analysis at specification time concentrate on those specifications being formulated using the model transformation language Henshin [3], and therefore can be formalized by algebraic graph transformation [12]. In this case, we can use critical pair analysis to discover several conflicts and dependencies between rules: *produce/use* (the first rule produces an element that is used by the second rule), *delete/forbid* (the first rule removes an element that is prohibited by the second rule), *produce/forbid* (the first rule produces an element that is prohibited by the second rule), and *delete/use* (the first rule removes an element that is used by the second rule).

For analyzing whether a model refactoring R is suitable to erase model smell S we have to check dependencies between rules in the smell specification (SC) and the initial check of the refactoring (IC). Since rules in SC are simple pattern rules (i.e. they do not alter the model) only exist/use-dependencies and exist/forbid-conflicts between rules in SC and IC can exist[10]. This leads to the following issues:

- *exist/forbid*: R **is not** suitable to erase S.
- *exist/use*: R **may be** suitable to erase S. R **is** suitable to erase S if for all rules ic in IC there exists a rule sc in SC so that ic can be embedded in sc.

For analyzing whether a model refactoring R can cause model smell S we have to check dependencies between rules in the model change specification of the refactoring (MC) and the smell specification (SC). Here, not only produce/use-dependencies and produce/forbid-conflicts can occur, but also delete/forbid-dependencies and delete/use-conflicts. This leads to the following issues:

- *produce/forbid*, *delete/use*: R **does not** cause S.
- *produce/use*, *delete/forbid*: R **may** cause S. R **causes** S if for all rules sc in SC there exists a rule mc in MC so that sc can be embedded in mc.

In summary, using a static analysis of smell respectively refactoring specifications formulated in Henshin, we can definitely state that (1) a model refactoring R **is not suitable** to erase model smell S, and (2) a model refactoring R **does not cause** model smell S in case of rule conflicts

4.5 Discussion of approaches

The interrelation analysis at application time calculates applicable refactorings for a set of given model elements. This can be helpful especially in cases in which no manual definition of relations between model smells and model refactorings are provided for a given meta-model. It helps the modelers because they do no longer have to think about the applicability of model refactorings themselves but will only be provided with those that can actually be applied. This approach, however, comes along with significant drawbacks.

The first (and most obvious) problem is that the applicability of a refactoring does not imply its usefulness. A manually defined relation is done by a designer with the definitive goal to erase a model smell using a given model refactoring. The analysis at application time, however, does not reflect any actual relation between the model smell occurrence (from which the analysis starts) and the refactorings found to be applicable. If a refactoring is found to be applicable, there is absolutely no guarantee that it will do any improvement on the model quality by erasing the smell. It simply means that the structure of a model will allow the application of the model refactoring on one of the model elements that is being part of the model smell occurrence. For example, renaming class *Assistant* in Figure 1 to *Foo* would not erase smell *Equal Attributes in Sibling Classes*, but it is an applicable refactoring in this context.

The second problem is that one might be stuck in a local optimum when listening to the guidance of the quantitative analysis of changes of model smell occurrences after applying a certain refactoring. For example, if the larger (composite) refactoring *Extract Superclass* is applied using the constituent refactorings *Insert Superclass*, *Pull Up Attribute*, and *Pull Up Operation*, the model becomes even more 'smelly' after the first refactoring step since model smell *Empty Superclass* is inserted. However, if refactoring *Extract Superclass* is explicitly specified and applied as composite refactoring (as provided by EMF Refactor; see Section 2.3), this effect does not occur.

Concerning the interrelation analysis at specification time, we implemented a first prototype that translates Henshin rules to AGG[11] and uses AGG's critical pair analysis module. First results show that this analysis approach

[10] These are specific kinds of produce/use-dependencies respectively exist/forbid-conflicts. Here, no model elements are produced by the first rule but their existence is required (as actually specified by rules in SC).

[11] http://tfs.cs.tu-berlin.de/agg

seems to be helpful in defining new manual relations between model smells and model refactorings respectively to verify existing ones by checking potential conflicts and dependencies.

In summary, it seems that a manual definition of relations between model smells and model refactorings would be the most advisable approach for proposing suitable refactorings to erase a specific model smell. However, both discussed analyses can help to set up new manual relations respectively to verify existing ones. Considering possibly inserted smells after refactoring application, the analysis at application time seems to be most suitable since the concrete modeling context is addressed (*What will exactly happen?*) whereas manual relations are likely set up in a more general way (*What can possibly happen?*).

5. Related work

In this section, we compare our tools and approaches to related ones in the areas EMF modeling, UML modeling, and smell respectively refactoring facilities for Java in Eclipse.

Since EMF has evolved to a well-known and widely used modeling technology, it is worthwhile to provide model quality assurance tools for this technology. The Epsilon language family[12] provides the Epsilon Validation Language (EVL)[13] to validate EMF-based models with respect to language-specific constraints. In their simplest form, EVL constraints are quite similar to OCL constraints. Furthermore, EVL also supports dependencies between constraints, customizable error messages to be displayed to the user and specification of fixes which users can invoke to repair inconsistencies. Since the purpose of EVL is to detect inconsistencies it can be used for model smell checking to a limited extent only. The quick-fixes are formulated in the Epsilon Object Language (EOL) which is the core language of Epsilon and therefore not specifically dedicated to model refactoring. Here, Epsilon provides the Epsilon Wizard Language (EWL) [15]. We compare our approach with EWL in [2]. In contrast to EWL, we provide a specification frame for refactorings which allows different specification mechanisms. Especially, we propose Henshin [3], a model transformation approach based on graph transformation concepts which supports more correctness checks than EWL. In contrast to EWL, we further use the LTK technology for homogeneous refactoring execution in Eclipse including e.g. a refactoring preview.

Another approach for EMF model refactoring is presented in [19][14]. Here, the authors propose the definition of EMF-based refactoring in a generic way, however do not consider the comprehensive specification of pre-conditions. Our experiences in refactoring specification show that it is mainly the pre-conditions that cannot be defined generically. (See [3] for a more complex refactoring with elaborated pre-condition checks.) Furthermore, there are no attempts to analyze EMF models wrt. model smell detection.

The MoDisco framework [9] provides a model-driven reverse engineering process for legacy systems in order to document, maintain, improve or migrate them. Here, several specific models are deduced (for example, Java models are deduced from Java code) which can be analyzed in order to detect anti-patterns and then be manually improved, for example by refactorings. The main difference between MoDisco and our tool suite is heavily substantiated by the intended purpose (reverse engineering vs. EMF modeling).

Considering UML modeling, quality assurance tools are not integrated in UML CASE tools, i.e. UML models have to be exchanged between modeling and assurance tools using their XMI representations. In the following, we give a rough overview on existing UML model quality assurance tools: Considering UML model refactoring, there is no mature tool support available yet. However, some research prototypes for model refactoring are discussed in the literature, e.g. in [11, 17, 18]. Most of them are no longer maintained. Furthermore, to the best of our knowledge no UML CASE tool provides capabilities to specify and detect UML model smells. Here, our tools can be integrated in EMF-based UML CASE tools like IBM's Rational Software Architect[15] to provide pre-defined smells and refactorings, or to specify new smells or refactorings depending on specific needs.

Code smells have been investigated within more than a decade. In [13] for example, Fowler presents his standard work on refactoring and code smells. He describes smells for object-oriented systems extensively and suggests refactorings to get rid of them. Furthermore, there are several tools available which can support code smell detection. For example, *Checkstyle*[16] automates the process of checking Java code wrt. specific coding standards, and *FindBugs*[17] uses static analysis to look for bugs in Java code. Both tools provide a quick-fix mechanism to handle found problems. However, they do not address potential impacts on further problems. Based on the Java Development Tools, a variety of Java code refactorings are provided in Eclipse using LTK. However, Eclipse does not offer an integration of code smells and code refactorings in the way we presented in this paper.

6. Conclusion

In this paper, we address the integration of model smells and model refactorings considering two directions. More specifically, we present mechanisms to support modelers in the selection of refactorings by (1) suggesting appropriate model refactorings in order to erase model smells, and (2) to get warnings in cases where new model smells occur due to applying a model refactoring. We discuss three alternative ap-

[12] http://www.eclipse.org/epsilon/

[13] http://www.eclipse.org/epsilon/doc/book/

[14] http://www.modelrefactoring.org/index.php/Refactoring

[15] http://www-01.ibm.com/software/awdtools/swarchitect/

[16] http://checkstyle.sourceforge.net/index.html

[17] http://findbugs.sourceforge.net/

proaches (manual integration and interrelation analysis both at application time and at specification time) and present the implementation of the first two approaches by extending the existing tools EMF Smell and EMF Refactor supporting smell detection and refactoring execution on models which are based on the Eclipse Modeling Framework.

As direction for future work we plan to substantiate our first reflections on the analysis of model smell and refactoring specifications which are based on graph transformation concepts (like specifications formulated in Henshin). Furthermore, it would be worthwhile to inspect whether further specification languages can be used for static analysis purposes. E.g., languages like EVL and EWL may also be integrated into our tools EMF Smell and EMF Refactor.

Besides tool optimization (e.g., grouping smells and refactorings to address scaling issues of the UI), we plan to integrate EMF Smell and the presented mechanisms into the open source Eclipse incubation project EMF Refactor in the near future. Doing so, we hope to address a broader audience with respect to using, maintaining, and evaluating our tools.

Acknowledgments

We like to thank the anonymous reviewers for their valuable comments on the previous version of this paper.

References

[1] S. W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2002. ISBN 0521525470.

[2] T. Arendt, F. Mantz, L. Schneider, and G. Taentzer. Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study. In *Model-Driven Software Evolution, Workshop Models and Evolution*, 2009. http://www.modse.fr/modsemccm09/doku.php?id=Proceedings.

[3] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In *MoDELS 2010*, LNCS, pages 121–135. Springer, 2010.

[4] T. Arendt, M. Burhenne, and G. Taentzer. Defining and Checking Model Smells: A Quality Assurance Task for Models based on the Eclipse Modeling Framework. In *9th edition of the BENEVOL workshop*, 2010. http://rmod.lille.inria.fr/benevol/pier.

[5] T. Arendt, F. Mantz, and G. Taentzer. EMF Refactor: Specification and Application of Model Refactorings within the Eclipse Modeling Framework. In *9th edition of the BENEVOL workshop*, 2010. http://rmod.lille.inria.fr/benevol/pier.

[6] T. Arendt, P. Stepien, and G. Taentzer. EMF Metrics: Specification and Calculation of Model Metrics within the Eclipse Modeling Framework. In *9th edition of the BENEVOL workshop*, 2010. http://rmod.lille.inria.fr/benevol/pier.

[7] T. Arendt, S. Kranz, F. Mantz, N. Regnat, and G. Taentzer. Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support. In *Software Engineering*, volume 183 of *LNI*, pages 63–74. GI, 2011.

[8] D. Astels. Refactoring with UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 67–70, Alghero, Italy, 2002.

[9] G. Barbier, H. Brunelière, F. Jouault, Y. Lennon, and F. Madiot. MoDisco, a Model-Driven Platform to Support Real Legacy Modernization Use Cases. In *Information Systems Transformation: Architecture-Driven Modernization Case Studies*, pages 365–400. The Morgan Kaufmann/OMG Press, 2010.

[10] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST*, 3, 2006. `http://easst.org/eceasst`.

[11] M. Boger, T. Sturm, and P. Fragemann. Refactoring Browser for UML. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 366–377. Springer, 2003.

[12] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.

[13] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, Reading/Massachusetts, 1999.

[14] M. Genero, M. Piattini, and C. Calero. A Survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4 (9):59 – 92, 2005.

[15] D. S. Kolovos, R. F. Paige, F. Polack, and L. M. Rose. Update transformations in the small with the Epsilon Wizard Language. *Journal of Obj. Tech.*, 6(9):53–69, 2007.

[16] C. F. Lange. *Assessing and Improving the Quality of Modeling: A series of Empirical Studies about the UML*. PhD thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, 2007.

[17] S. Markovic and T. Baar. Refactoring OCL Annotated UML Class Diagrams. *Software and Systems Modeling*, 7:25–47, 2008.

[18] I. Porres. Model Refactorings as Rule-Based Update Transformations. In *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 159–174. Springer, 2003.

[19] J. Reimann, M. Seifert, and U. Aßmann. Role-Based Generic Model Refactoring. In *Model Driven Engineering Languages and Systems, 13th International Conference, MoDELS 2010. Proceedings*, LNCS, pages 78–92. Springer, 2010.

[20] D. Steinberg, F. Budinsky, M. Patenostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison Wesley, 2008.

[21] G. Sunyé, D. Pollet, Y. Le Traon, and J. Jézéquel. Refactoring UML models. In *UML 2001: 4th International Conference on the Unified Modeling Language*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.