Refactoring UML models

Sebastian Geiger and Kristof Meixner

[1] `sbastig@gmx.at`
Registration No. 1127054
[2] `kristof.meixner@fatlenny.net`
Registration No. 9725208

Abstract. Over the last twenty years refactoring advanced to a commonly known and used techniques in modern software engineering. We present an overview from the beginning of refactoring in source code to its actual application in model-driven software development. Furthermore we discuss methods that ensure that refactored models are still correct after modifications like static and dynamic analysis. We present research on a subset of UML that can be executed and thus tested dynamically. At the end we introduce some works that we will adapt in our project work to implement refactorings that are done on class diagrams and impact corresponding activity diagrams.

Table of Contents

1 Introduction

Model-based software development or model-driven software development is not only an extensive field of research but receives also more and more attention from the industry. Nowadays models are not only used as visual explanations of software concepts but as source for the development process itself. Thus models need to provide an abstraction of the represented concepts in high quality. To provide quality assured software artifacts that last over the software lifecycle a technique called refactoring is often used to restructure in our case models. Refactorings should improve the quality and also the understandability of the models as well as adapt the models to changes coming from from the domain. The most important requirement for refactorings is that the preserve the behavior of the model. In our project work we focus on an set of refactorings that should be implemented taking the correlations between different model types as well as static and dynamic analysis into account. In this work we concentrate on the related work in a broader sense and present them in a chronological manner.

The paper is structured as follows. Section 2 copes with the early steps of refactoring in the domain of source code development. The following section discusses how commonly known refactorings from source code where adapted in theoretical works to models.

2 Beginning in Refactoring

In this section we will present an overview on the beginnings in structured refactoring of source code. In his thesis [12] Opdyke "defines a set of program restructuring operations" that "preserve the behavior of a program" to increase software quality. This technique became known as refactoring.

The motivation of his work is on the one hand that software should be reused because of the high costs of development and on the other hand that software needs to be restructured over its lifecycle to maintain this reusability. The issue he addressed in his work is the problem of changing parts of source code from an object oriented system, grounded in a possibly large code base while also maintaining all the references and dependencies manually. He described this process as "time consuming, difficult and error prone". As a solution he proposes "an approach for providing automated support for the restructuring", plans to reorganize the source code on an intermediate level without changing the behavior of the program.

A section in the thesis covers behavior preserving approaches. In this chapter he first mentions the usage as "preconditions" to ensure that a refactoring does not corrupt the syntax and more important does not change the execution behavior of the program. He quickly refuses the approach to use the static compiler to test these issues as the compiler is not able to catch "some errors that could change the behavior of the program". In this section also some rules are presented that can be tested beforehand to check if a refactoring is possible on the concepts. A rule to highlight copes with semantic preservation of code and

is described as "Semantically Equivalent References and Operation". He defines that semantic equivalence as follows: *"let the external interface to the program be via the function main. If the function main is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same"*. This means that from outside of a specific system circle the refactoring is not visible. The range of this circle however differs from the impact of the refactoring on the system. In the rest of his work he shows some examples of refactorings depending on their level of implication for the source code for *Smalltalk* program concepts and how they could be applied.

Roberts [14], writing his thesis on the same University, builds large parts of his contribution on the work of Opdyke. He criticizes that the refactorings described by Opdyke are to small and "are rarely performed in isolation" but in a sequence steps to refactor to a better "design in mind". He also mentions that the "expensive analysis" for legal code after a refactoring should be eliminated and introduces postconditions along with Opdykes preconditions to guarantee behavior preservation. Besides his contribution of "a formal definition of refactoring that is based on postconditions" Roberts presents a dependency definition of refactorings that is based on the commutativity of single changed put in sequence. Furthermore issues an idea, which in our work seems rather interesting, to analyze very complex refactorings not in a static manner but dynamically via refactoring during execution. This in particular fits to our use case in the project that builds on executing and tracing changes due to refactoring before and after the modification. Roberts also redefines the refactorings developed by Opdyke and presents his catalog that can be used as a source for modern refactoring.

Another extensive catalog of refactorings based on source code modifications in *Java* was developed and described by Fowler in [3]. His list of refactorings[3] is continuously updated and extended for different programming language and shows also good examples how to refactor in design pattern thinking.

3 From source code to model refactoring

In the last section we gave a brief introduction to the early research in refactoring. In this section we will discuss what model refactoring is and how it benefit from from source code refactoring. As already mentioned in Section 1 modern software development more and more uses the concepts of model-based and model-driven approaches which means that formal models are one of the most important artifacts in development. This also implies that changes in the domain of the software have to be propagated to the formal models. With faster development cycles common in agile software development like eXtreme programming ([2]) or Scrum ([13]) those changes have to be done even more efficient and reliable. It is obvious that in this case model development benefits from the techniques of refactoring introduced in the late nineties to source code. However models need to be treated a little different than source code.

---

[3] The catalog can be found online under http://refactoring.com/catalog/
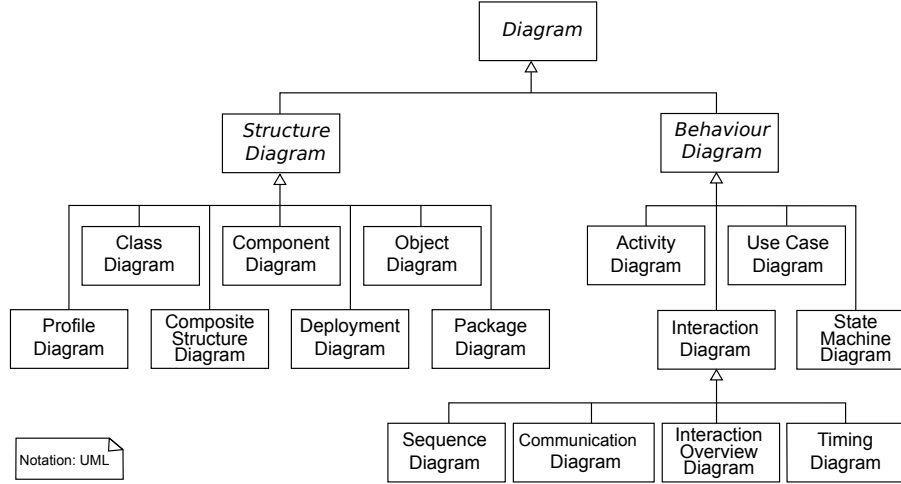
Fig. 1. *UML* diagram type hierarchy (Derfel73, PMerson)

Today the de-facto standard the *Unified Modeling Language* (*UML* - [10]) standardized by the *OMG*[4] in 1997 is a general purpose language used for model creation and development. *UML* consists of an abstract syntax which builds the foundations for the models which are formulated in the concrete syntax of the language. *UML* defines several different diagram types that can be used for designing parts of a software system. The most commonly diagram type is the class diagram which is often used to depict models and components of an architecture. Generally the diagrams can be distinguished into structural and behavioral diagrams. A further partitioning of the diagrams is shown in Figure 1. The different diagram share a common model in the background that is often serialized in *XMI*[5] an *XML* exchange format which is also standardized by the *OMG*.

The *OMG* furthermore defined the *Object Constraint Language* (*OCL* - [9]) which is "a formal language used to describe expressions on *UML* models" [9]. The expressions specify mainly invariants but also preconditions and postconditions that have to hold for the modeled system.

One of the first works in the domain of model refactoring was done by Sunyé et al. in [15]. They state that refactoring of models is difficult because "it is often hard to measure the actual impact of modifications" and is "particularly true for the Unified Modeling Language, with its various structural and dynamic views, which can share many modeling elements". Sunyé et al. brings up an example of a class diagram and a state chart diagram and describes how some refactorings can be applied on these models which are backed up be pre- and postconditions

---

[4] http://www.omg.org/
[5] http://www.omg.org/spec/XMI/

3

formulated in *OCL* to preserve the behavior of the models. The work gives a good idea how model refactorings can be achieved and statically tested. However also the authors state that "the search for some UML specific refactorings has been somehow frustrating, specially when we wanted transformations to have an impact on different UML views". We also somehow experienced that most of the reviewed research concentrated only on a single type of diagram in their work.

Gorp et al. [4] in addition to the work before contributed a proposal for the extension of *UML* in 2003 and the use of code smells in combination with *OCL* pre- and postconditions to bundle so called "refactoring contracts". Code smells are symptoms for design flaws and might indicate problems that are hidden deeper in the source code. Gorp recommended to use a threefold process for refactoring. First use preconditions to verify that a refactoring is possible, second use a code smell detector to find smells that indicate that the model does not benefit from a refactoring and last but not least use the postconditions for a final check. In their research they adapted the method for two refactorings namely *pullup method* and *extract method*.

Markovic and Baar in [5] presented a catalog of refactorings that leans on the catalog of Fowler and classify them by their impact on the meta-model representation of the models. Simple refactorings might be easy to track and impacts may be traced and addressed on the fly however if the meta-model representation changes the refactorings can result in complex changes of the model and the corresponding diagram types. They therefore formalize their refactoring rules in a graphical notation included in the QVT^6 standard by the *OMG*. They also present an implementation of the refactorings in *Eclipse*[7].

The last research we discuss in this section is the one of Arendt and Taentzer [1]. Arendt and Taentzer relied on the work of Mohagheghi et al. [8] that condensed six classes of quality goals in model development out of several studies. With these goals in mind they formulated a set of questions that can be checked for specific models and show possible code smells in the underlying models. Furthermore they implemented a complete framework based on the *Eclipse Modeling Framework*[8] called *EMF Refactor*[9]. This plugin to *Eclipse* allows to calculate defined model metrics and model smells for a given model. The user can then choose from various refactorings how to improve the model and make it more understandable.

In our project work we chose to use this framework for our own implementation because it can be easily extended and supports a wide range of technologies that can be used to implement the refactorings. Besides the already mentioned *OCL* additional modules can be created in *Java, Henshin*[10] a transformation engine for *EMF* and *ComRel* "a model-based language for the combination of EMF model refactorings".

---

[6] http://www.omg.org/spec/QVT/

[7] http://www.eclipse.org/

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

[9] http://www.eclipse.org/emf-refactor/

[10] https://www.eclipse.org/henshin/

4 From static to dynamic testing

In the section above we presented some works that cover refactoring of models and use different methods of static analysis to ensure that the behavior of the models are preserved and to identify models that where not well designed. In this section we will discuss research that adapts the approach presented by [14] and uses the execution of systems to dynamically verify the behavior of models.

UML does not provide the semantics to make a model as executable as program code is. This issue was issued in the definition of a subset of *UML* that narrows the semantics of the concepts down to make them executable. This subset is defined in the standard "Semantics of a Foundational Subset for Executable UML Models" (*fUML* - [11]). If you recall Figure 1 *fUML* supports two diagram types, the class diagram and the activity diagram. The class diagram can be used to model the domain with its concepts, the activity diagram can be used to describe the behavior of the concepts in the class diagram. The abstract syntax of *fUML* differs slightly from the one of *UML*.
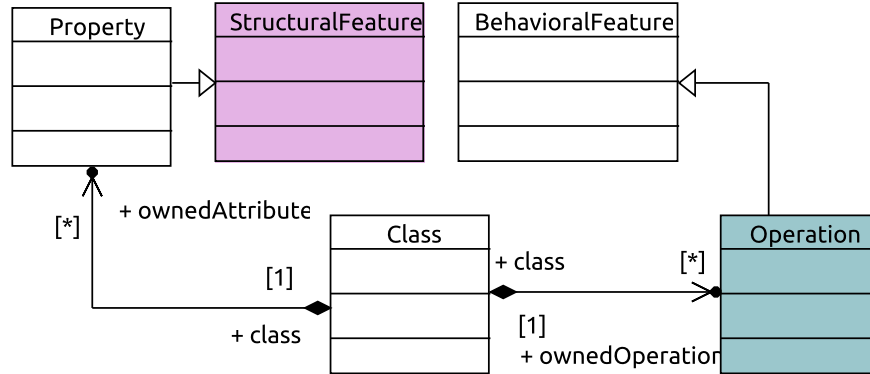
Fig. 2. Abstract syntax for used class concepts in *fUML*

In our project work we concentrate on this subset to investigate the changes that are implied in the corresponding diagram if the other diagram changes. For this work we only apply refactorings on the class diagram and observe the changes implied in the activity diagram as this to be the more reasonable use case. In order for the reader to get more insight on the topic we provide a simple *encapsulate field* example refactoring. The refactoring has the following steps:

- Create setters and getters for the attribute.
- Change all occurrences of the attribute to the created setters and getters.
- Change the attribute to private.
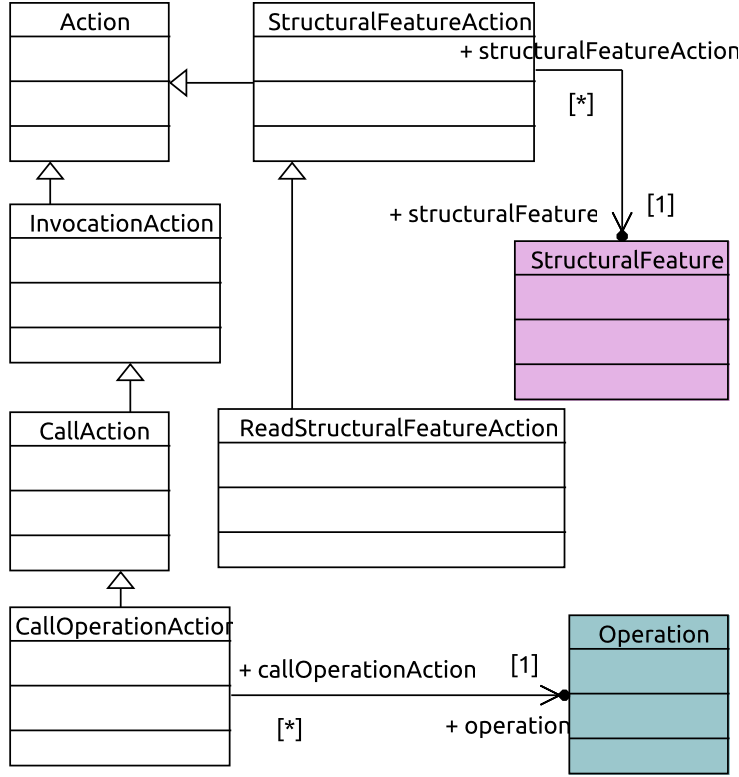
Fig. 3. Abstract syntax for used actions in *fUML*

In Figure 2 and 3 the abstract classes for the class diagram and the activity diagram are displayed. For the class diagram there are nearly no modifications except to create the new methods and set the attribute to private. In an activity diagram it is not that obvious. Figure 4 shows a very simple activity diagram. Each of the actions readNumberOfVehicles, readCustomer and readAge read StructuralFeatures shown in violet in the figures are actions of type StructuralFeatureAction. Each of these actions have to be changed to CallOperationActions as the accessed parts of the class diagram changed to operations shown in turquoise. In this simple model it seems rather easy but in complex models dependencies grow and make necessary changes it hard to target.

Mayerhofer proposes in [6] to use common techniques from software development to dynamically verify the behavior of refactored models namely testing and debugging. However some problems come up when taking a deeper look because "UML models constitute a multiple view specification of a system". In a further scientific work by Mayerhofer et al. [7] a runtime model is presented that
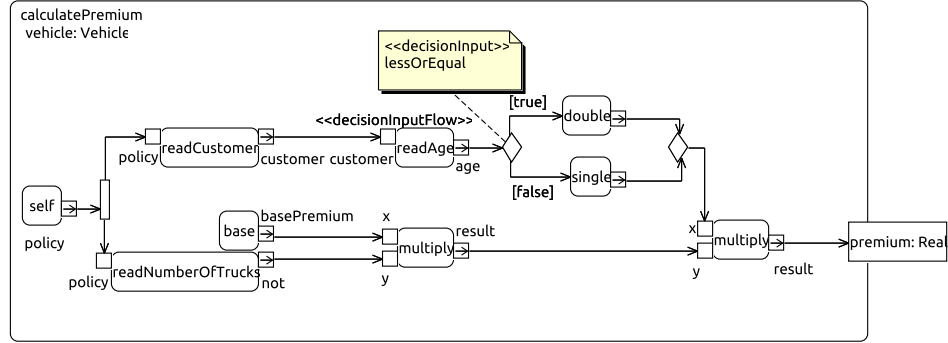
Fig. 4. Calculate premium activity diagram with refactorings

is capable to execute and trace activities. They also implemented[11] a virtual machine that is capable of running *fUML* models and can even convert *UML* model compatible to *fUML* for execution purposes.

For our project work we use this reference work to implement a set of refactorings test the pre- and postconditions on them and run them in the virtual machine after refactoring to verify the preservation of their initial behavior.

5 Conclusion

We presented a brief overview on the scientific work in the domain of refactoring over the last twenty years. It ranges from simple source code refactoring with pre- and postconditions and developed to the domain of model-driven development where refactoring is equally needed but harder to implement because of the different views on the underlying models and their correlations. Model development tries to take common techniques from source code development and adapt them for its needs. While from the beginning static analysis was taken into account in the later research and especially with the standardization of *fUML* also dynamic analysis like testing and debugging during execution is introduced.

For our project work we are going to use a combined approach and use static as well as dynamic model testing to implement a set of refactorings for class and activity diagrams.

---

[11] http://www.modelexecution.org/

7

References

1. ARENDT, T., TAENTZER, G., AND WEBER, A. Quality assurance of textual models within eclipse using ocl and model transformations. In *OCL@MoDELS* (2013), pp. 1–12.
2. BECK, K. Embracing change with extreme programming. *IEEE Computer 32*, 10 (1999), 70–77.
3. FOWLER, M. *Refactoring - Improving the Design of Existing Code*. AddisonWesley, July 1999.
4. GORP, P., STENTEN, H., MENS, T., AND DEMEYER, S. Towards automating source-consistent uml refactorings. In *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, P. Stevens, J. Whittle, and G. Booch, Eds., vol. 2863 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 144–158.
5. MARKOVIC, S., AND BAAR, T. Refactoring ocl annotated uml class diagrams. *Software and System Modeling 7*, 1 (2008), 25–47.
6. MAYERHOFER, T. Testing and debugging uml models based on fuml. In *ICSE* (2012), pp. 1579–1582.
7. MAYERHOFER, T., LANGER, P., AND KAPPEL, G. A runtime model for fuml. In *Models@run.time* (2012), pp. 53–58.
8. MOHAGHEGHI, P., DEHLEN, V., AND NEPLE, T. Definitions and approaches to model quality in model-based software development - a review of literature. *Information & Software Technology 51*, 12 (2009), 1646–1669.
9. OMG. *OMG Object Constraint Language*, 2.3.1 ed. OMG, http://www.omg.org/spec/OCL/2.3.1, 01 2011.
10. OMG. *OMG Unified Modeling Language*, 2.4.1 ed. OMG, http://www.omg.org/spec/UML/2.4.1/, 05 2011.
11. OMG. *Semantics of a Foundational Subset for Executable UML Models*, 1.1 ed. OMG, http://www.omg.org/spec/FUML/1.1, 08 2013.
12. OPDYKE, W. F. Refactoring object-oriented frameworks. Master's thesis, University of Illinois, 1992.
13. RISING, L., AND JANOFF, N. S. The scrum software development process for small teams. *IEEE Software 17*, 4 (2000), 26–32.
14. ROBERTS, D. Practical analysis for refactoring. Master's thesis, University of Illinois, 1999.
15. SUNYÉ, G., POLLET, D., TRAON, Y. L., AND JÉZÉQUEL, J.-M. Refactoring uml models. In *UML* (2001), pp. 134–148.