fUML Refactoring with EMF[⋆]

Sebastian Geiger (1127054) and Kristof Meixner (9725208)

Business Informatics Group
Vienna Technical University

Abstract. In this work we will present some ideas and concepts for the refactoring of fUML. The main contribution of this work is the extension of existing UML refactorings to cover not only the static aspect of UML such as class diagrams but to also include refactorings for dynamic parts such as activity diagrams. In this work we will present basic concepts for refactoring with EMF and show how model semantics can be preserved through the use of OCL constraints. In the remainder of the paper we then present our tool chain and the used technologies such as EMF and Ecore and how we used them for refactoring. We also present a discussion of EMF.Refactor, which shows how such refactorings can be made available in the Eclipse GUIs such as the EMF tree editor or Papyrus.

[⋆] This work has been created in the context of the course "Advanced Model Engineering" (188.952) in SS14.

Table of Contents

1 Introduction 1
2 Motivation 2
3 Refactorings Examples 2
4 Refactoring of fUML diagrams 3
5 Tool chain and implementation 4
  5.1 Model refactoring 4
  5.2 GUI Integration 5
6 Related Works 5
7 Conclusion 5
References 5

1 Introduction

Model-based software development or model-driven software development is not only an extensive field of research but gets also more and more attention from the industry side. Nowadays models are not only used as visual explanations of the underlying concepts but as source for the development process itself. Thus models need to provide an abstraction of the represented domains in a high quality. Mohagheghi et al. [6] discussed possible quality attributes of models in their work.

To represent formal models the OMG developed the UML [8] which by now advanced to an industry standard for modeling. With these common semantics enriched models can be preserved over time and even reused.

Nevertheless models might have to be revised over the lifecycle of the software. With todays trend to more agile software development such as eXtreme Programming [2] or Scrum [11] changes on models have to be even more efficient which brings refactoring of models into account.

Refactoring is a technique that originates from source code development but can also be applied to model engineering. The goal is to introduce behavior preserving changes [10] that increase the quality and understandability of the models.

While refactoring source code respectively textual code applies to a single type of represenation in UML different representations of models exist as various diagram types. This makes behavior preserving refactoring even harder as it needs to span over those different types of diagrams and semantics.

To prove the correctness of models before and after refactoring different approachs exist. One is the static analysis of UML models via metrics and the attempt to find model smells [1] which can be done via OCL [7]. Another one is to verify that models can be executed and debugged in some way. The OMG introduced a subset of UML named fUML [9] which narrowly defines semantics for class and acivity diagrams to make them executable. Mayerhofer [5] in her work furthermore proposed a framework based on fUML that is able to execute and debug those models.

The goal of this work is to introduce refactorings for fUML models, examine the requirements for co-refactoring of the corresponding diagram types and define which co-changes have to be done to preserve the behavior. Our approach to prove the correctness of the models after refactoring is twofold. On the one hand side we use pre- and postconditions [12] to define if the specific changes can be applied and that the models are semantically correct afterwards. On the other hand side the models have to stay executable as well as traceable after the changes.

The rest of the work is structured as follows. In section X we give an overview over fUML and provide a motivating example of a model that is used throughout the paper. Section Y describes a selection of useful refactorings inspired by Fowler [3] and Markovic and Baar [4] and their effects on class diagrams as well as activity diagrams. In section Z we show which pre- and postcondition that are needed for the refactorings and how we refactor the models. In section AA we describe the toolchain that we use to define the models, implement a set of refactorings and test them. Related work is covered in section AC and a conclusion is drawn to summarize the paper.

2 Motivation

fUML is an extension of UML which builds on a subset of UML classes with the purpose of adding semantics to UML models such that they can be executed on the model level. The dominant concept for this is the activity diagram. If a refactoring is performed on a UML model such as a class diagram, then any activity diagram which is releated to the class diagram, has to be checked and possibly changed as well. In section 4 we will present some examples of fUML activity diagrams and present the implications that result from changing class diagrams.

Since a refactoring changes the structure of a model it is important to ensure that all changes maintain the original semantics of the model. Violating this requirement can result in models with either a different behavior, or in models which can no longer be executed. To ensure semantic preservation, two main techniques can be used. First, the refactoring can be broken down into smaller steps, each of which either guarantees to preseve the semantics of the model or makes it easier to verify that this is the case. Second, logical constraints can be used to limit refactorings on models to only those cases where semantic preservation can be ensured. For this purpose pre- and postconditions are specified with OCL constraints. A refactoring is then only applied if the original model satisfies the precondition before the refactoring is applied and the postcondition after the refactoring has been completed. Such constrains must be individually specified for each kind of refactoring that is to be performed, as such a part of this paper will discuss different OCL constraints for the refactorings that we introduce.

In this paper we will present a set of refactorings and give examples of how each refactoring can be applied to a concrete model.

3 Refactorings Examples

This section covers some general refactorings of UML class diagrams. As the basis of these refactorings an example from the insurance domain is used. In the insurance business there are domain objects such as an insurance policy. Cars and trucks can be insured by adding them to the policy. There is an insurance company which has customers and employees and employees may purchase an insurance policy for one of their cars or trucks. Figure 1 shows a class diagram of such an insurance policy. This class diagram would benefit from several possible refactorings such as an *extract superclass* which can be applied to both Car and Truck to extract a Vehicle class. A simple *extract class* can be used on InsurancePolicy to extract the from and until dates into their own InsurancePeriod class. As part of the *extract superclass* refactoring two additional refactorings namely *pull up attribute* and *pull up method* are used to move the identical attributes and methods of both classes to the new superclass. As the attributes weight and registration are public, we can use *encapsulate field* to make them private and provide getter and setter implementations. Finally the method addCar can be renamed into addVehicle with *rename method* and the addTruck method can be removed with *remove method*.

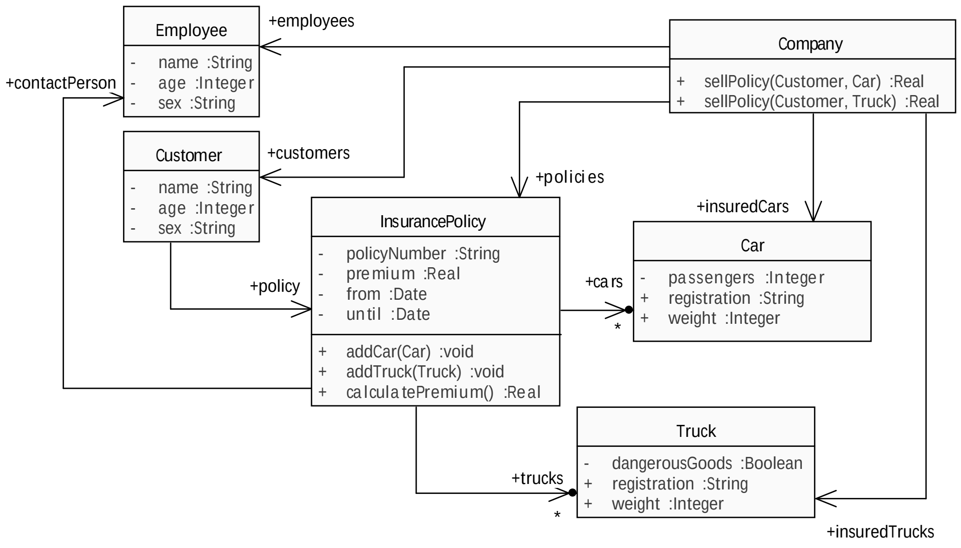


Fig. 1. A car object that could benefit of extract class

For each of the methods the *InsurancePolicy* and *Company* classes a separate activity diagram exists which defines the semantics of these methods. As some of them are quite similar we will present only the diagrams for addCar(), sellPolicy(Customer, Car) and calculatePremium().

Figure 4 shows an example of a more complex car object which captures additinoal information about the owner which is allowed to drive the car. This example would benefit from an *extract class* refactoring. The extract class refactoring would pull the members The policy allows to insure cars and trucks. Both Car and Truck have some attributes that are similar (weight, registration) and others that are different (dangerousGoods, passengers). From the

In this section we will present some general refactorings such as the “extract superclass” refactoring. A list of example refactorings is presented in figure 3.

4 Refactoring of fUML diagrams

In this section we will present some general refactorings

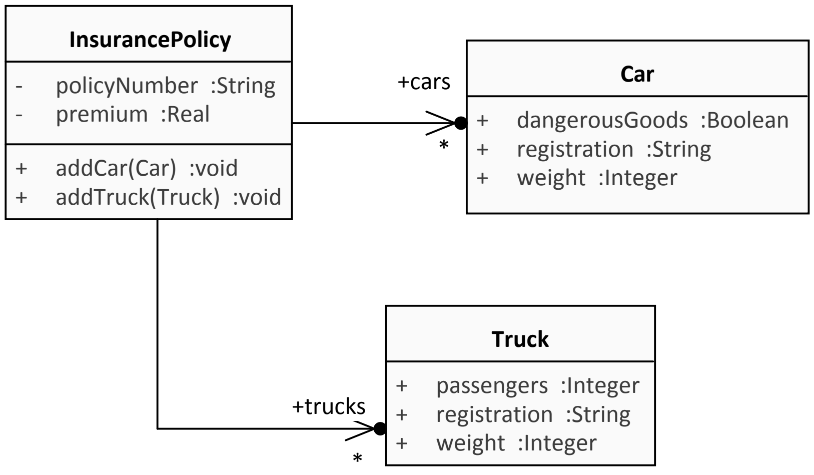


Fig. 2. Insurance policy class diagram

Extract class
Extract superclass
Rename Class
Rename Method
Rename Variable
Add / Remove Parameter
Encapsulate Field
Pull up attribute
Pull up operation
Pull up association end
Remove unused class

Fig. 3. Refactoring examples

5 Tool chain and implementation

For our tool chain we have relied on the moliz[1] repository, mainly for the ability to execute the fUML models with a virtual machine. The models are stored as XMI

5.1 Model refactoring

Describe our tool chain, how we created models, how we load them, what information of the abstract syntax we use for refactoring, etc.

[1] source...

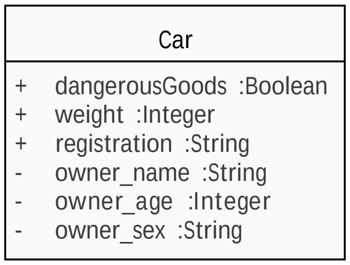
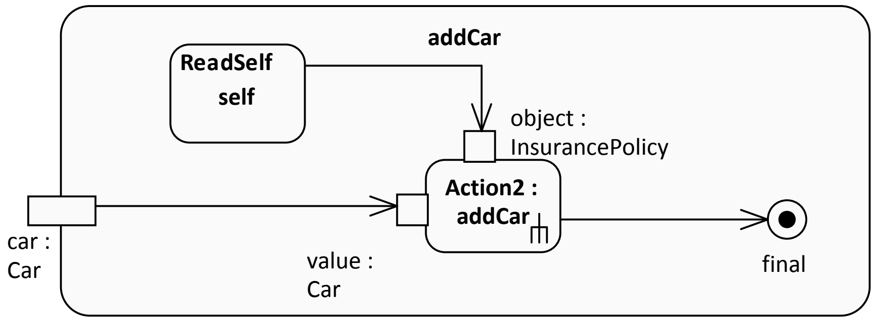


Fig. 4. A car object that could benefit of extract class



5.2 GUI Integration

Describe what we did with EMF.Refactor.

6 Related Works

We have compared our works with several other available papers. In [..] there is a discussion of uml refactings which covers ....

some related works such as ...

[12]

7 Conclusion

We conclude this paper with...

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. Markovic, S., and Baar, T. Refactoring ocl annotated uml class diagrams. *Software and System Modeling 7*, 1 (2008), 25–47.
5. Mayerhofer, T. Testing and debugging uml models based on fuml. In *ICSE* (2012), pp. 1579–1582.
6. 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.
7. OMG. *OMG Object Constraint Language*, 2.3.1 ed. OMG, http://www.omg.org/spec/OCL/2.3.1, 01 2011.
8. OMG. *OMG Unified Modeling Language*, 2.4.1 ed. OMG, http://www.omg.org/spec/UML/2.4.1/, 05 2011.
9. OMG. *Semantics of a Foundational Subset for Executable UML Models*, 1.1 ed. OMG, http://www.omg.org/spec/FUML/1.1, 08 2013.
10. Opdyke, W. F. Refactoring object-oriented frameworks. Master's thesis, University of Illinois, 1992.
11. Rising, L., and Janoff, N. S. The scrum software development process for small teams. *IEEE Software 17*, 4 (2000), 26–32.
12. Roberts, D. Practical analysis for refactoring. Master's thesis, University of Illinois, 1999.