# 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

Todays software development processes focus more and more on the models than on designing source code. Models are not only used as abstract visual explanations of the underlying concepts but as the origin for source code generation. The *Unified Modelling Language* advanced to a standard for designing models for software systems on an abstract level. The concept of refactoring is an important part of software development which affects all artifacts of the design process and should be included in every cycle of iterative and evolutionary software development. It main goal is to reorganise software components in a behavior preserving way [3] to achieve better quality.

The quality of the models and thus refactoring impacts can be either measured in a static way by *code smells* or *code metrics* as proposed by Aendt et.al. [1] or in a dynamic way by *executing* and *debugging* as proposed by Mayerhofer [2].

(The concept of refactoring is an important part of software development which should be included in every cycle of iterative and evolutionary software development. It main goal is to reorganise software components to achieve better design, lower coupling, higher cohesion, and other attributes which make up good software design. The important aspect of refactoring is that no additional features are introduced during a refactoring step and the semantics of the software are maintained; no additional bugs or errors are introduced. Refactoring can not only be applied to source code but as well to models such as UML or fUML models. In the context of models such as class diagrams, it means that the structure of the diagram is rearranged to achieve a better design. This included changing superclass releationships, moving attributes and methods to different classes or adding new interfaced. A catalogue of possible UML refactorings can be found in .)

cite markovic

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.

## 2 Motivation

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*.

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.

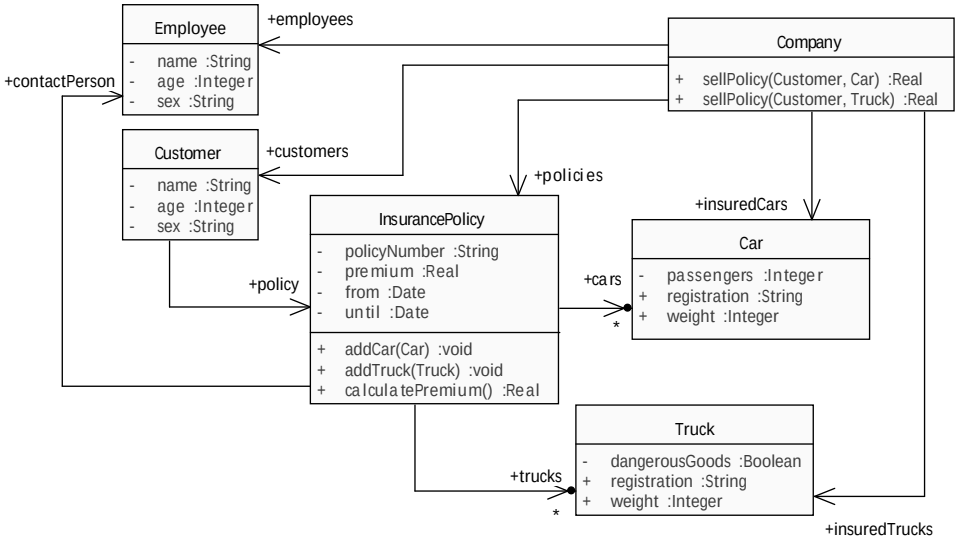


**Fig. 1.** A car object that could benefit of extract class

## 4 Refactoring of fUML diagrams

In this section we will present some general refactorings

## 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.

### 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 ...

[4]

[1] source...

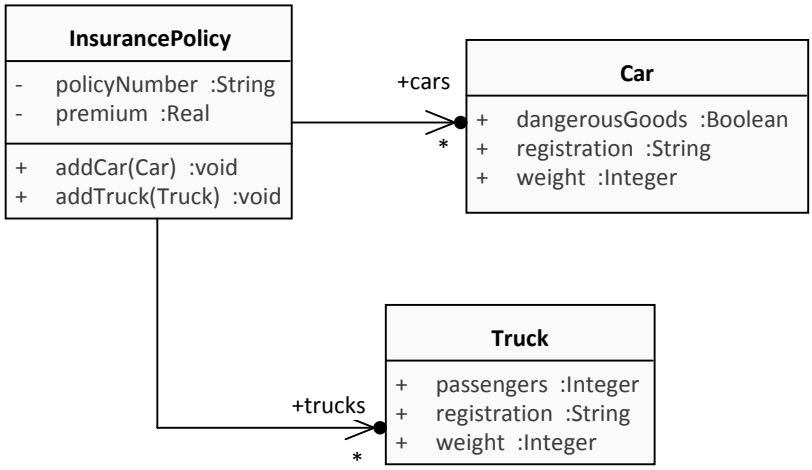


**Fig. 2.** Insurance policy class diagram

| |
|---|
| 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

## 7 Conclusion

We conclude this paper with...

## References

1. Arendt, T., and Taentzer, G. A tool environment for quality assurance based on the eclipse modeling framework. *Autom. Softw. Eng. 20*, 2 (2013), 141–184.
2. Mayerhofer, T. Testing and debugging uml models based on fuml. In *ICSE* (2012), pp. 1579–1582.
3. Opdyke, W. F. Refactoring object-oriented frameworks. Master's thesis, University of Illinois, 1992.
4. Roberts, D. Practical analysis for refactoring. Master's thesis, University of Illinois, 1999.

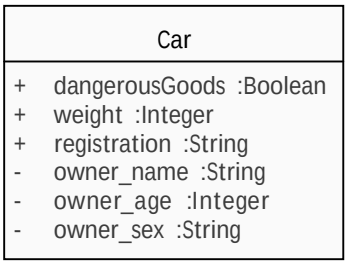


**Fig. 4.** A car object that could benefit of extract class