

fUML Refactoring with EMF

Sebastian Geiger (1127054) and Kristof Meixner (9725208)

Business Informatics Group
Vienna Technical University

Abstract. In this work we will present ideas and concepts for the refactoring of fUML models. 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 of models 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 toolchain and the used technologies of EMF (in particular Ecore and OCL) and how we used them for refactoring of models. We also present a discussion of EMF.Refactor, which shows how such refactorings can be made available in the Eclipse GUIs such as the UML tree editor or Papyrus.

Table of Contents

1	Introduction.....	1
2	Motivation	2
3	Refactoring Examples	2
4	Refactoring of fUML models.....	6
4.1	Rename refacorings	7
4.2	Extract superClass	7
4.3	Pull up attribute & pull up method.....	7
4.4	Encapsulate attribute	8
5	Toolchain and implementation.....	8
5.1	Model refactoring	9
5.2	GUI Integration	10
6	Current Limitations	10
7	Related Work	10
8	Conclusion	11
	References	12

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 domains in a high quality. Mohagheghi et al. [11] discussed possible quality attributes of models in their work.

To represent formal models the OMG developed UML [13] 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 today's trend to more agile software development such as eXtreme Programming [4] or Scrum [16] changes on models have to be even more efficient which brings refactoring of models into the focus of research.

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 [15] that increase the quality and understandability of the models.

While refactoring source code and textual code respectively applies to a single type of representation, in UML different types of diagrams exist to represent various aspects of the models. 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 approaches exist. One is the static analysis of UML models via metrics and the attempt to find model smells [2] which can be done via OCL [12]. Another way is to verify that models can be executed and to compare the execution traces of the original and refactored models. The OMG introduced a subset of UML named fUML [14] which precisely defines semantics for class and activity diagrams to make them executable. Furthermore in [9] Mayerhofer 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 performed to preserve the behavior. Our approach to verify the correctness of the models after refactoring is twofold. On the one hand we use pre- and postconditions [17] to define if the specific changes can be applied and that the models are semantically correct afterwards. On the other hand the models have to stay executable as well as traceable after the changes.

The rest of the work is structured as follows. In section 3 we give an overview over fUML and provide a motivating example of a model that is used throughout the paper. Section 4 describes a selection of useful refactorings inspired by Fowler [6] and Markovic and Baar [8] and their effects on class diagrams as well as activity diagrams. In section 5 we show which pre- and postcondition are needed for the refactorings and how we refactor the models. In section 6 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 7 and a conclusion is drawn in section 8 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 related 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 (a procedure called co-refactoring).

so far we have not
done that

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 preserve 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 constraints must be individually specified for each kind of refactoring that is to be performed. In this paper we will introduce and discuss different OCL constraints for the refactorings that we introduce.

Refactorings are only useful if they can be easily applied to the models through an easy to use process such as a graphical front end. EMF.Refactor allows the integration of refactorings into editors such as the UML tree editor or papyrus. It provides a Java API as well as a module concept to integrate refactorings. We will discuss the use of EMF.Refactor as part of our tool chain presentation.

3 Refactoring Examples

This section covers some refactorings of UML models as well as a model which we use as an example for this work. Markovic [?] presented a list of refactorings which is relevant for changes in combination with class diagrams and OCL constraints. We use this catalog as an input and adapt it for our use case. The resulting list of refactorings which we plan to evaluate and implement is shown in Figure 1.

As the basis for these refactorings an example from the insurance domain is used. In the insurance business there are domain objects such as an insurance

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

Fig. 1. Refactoring examples

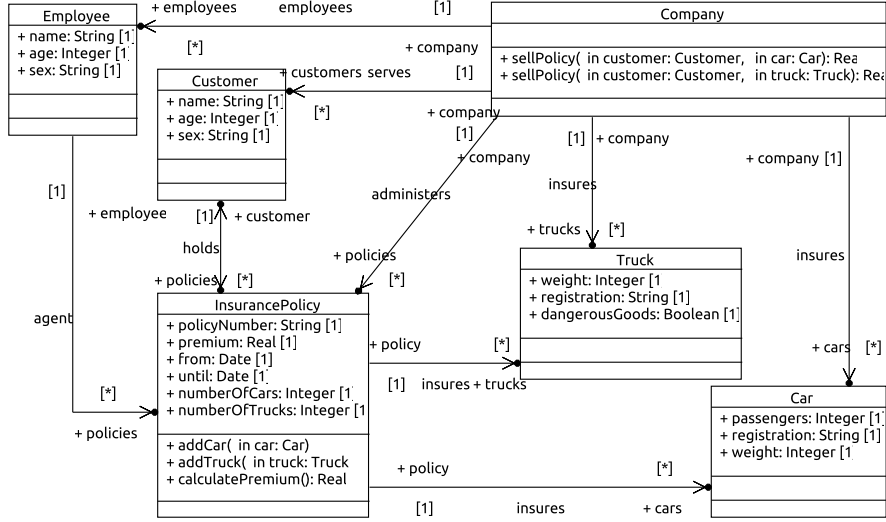


Fig. 2. Insurance class diagram without refactorings

policy. Cars and trucks can be insured by adding them to the policy. There is an insurance company which has customers and employees where customers may purchase an insurance policy for their cars or trucks. Figure 2 shows a class diagram of such an insurance company. 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

-
- ```

 graph LR
 self((self)) -- policy --> Join1(())
 Join1 --> readCustomer[readCustomer]
 readCustomer -- customer --> readAge[readAge]
 readAge -- age --> Decision1{ }
 Note1["<<decisionInput>>
lessOrEqual"] -.-> Decision1
 Note2["<<decisionInputFlow>>"] -.-> Decision1
 Decision1 -- [true] --> double[double]
 Decision1 -- [false] --> single[single]
 double --> Join2(())
 single --> Join2
 Join1 -- policy --> readCars[readNumberOfCars]
 readCars -- noc --> base[base]
 Join1 -- policy --> readTrucks[readNumberOfTrucks]
 readTrucks -- not --> add[add]
 base -- base --> multiply1[multiply]
 add -- addedUp --> multiply1
 multiply1 -- result --> multiply2[multiply]
 Join2 --> multiply2
 multiply2 -- premium --> output[premium: Real]

```
- The diagram illustrates the logic for calculating a premium. It starts with a 'self' node leading to a join node. From the join, the flow splits into two parallel paths. The top path reads 'customer' data (readCustomer) and 'age' (readAge), which then enters a decision diamond. A decision input 'lessOrEqual' is provided to the diamond. If the condition is true, the flow goes to a 'double' node; if false, it goes to a 'single' node. Both paths then merge at another join node. The bottom path from the initial join reads 'policy' data (readNumberOfCars and readNumberOfTrucks). These lead to 'base' and 'add' nodes respectively, which then merge and enter a 'multiply' node. The result of this multiplication is then multiplied by the value from the top path's join node at a final 'multiply' node, resulting in the 'premium: Real' output.

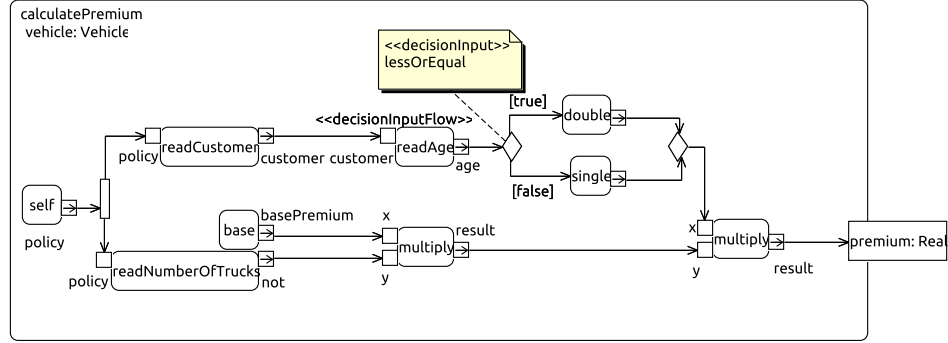
```

stateDiagram-v2
 [*] --> self
 self --> policy1 : policy
 policy1 --> read :
 read --> add : noc
 add --> write : oneAdded
 write --> join : [true]
 join --> [*] : [true]
 car --> addCar : car
 addCar --> join : [true]

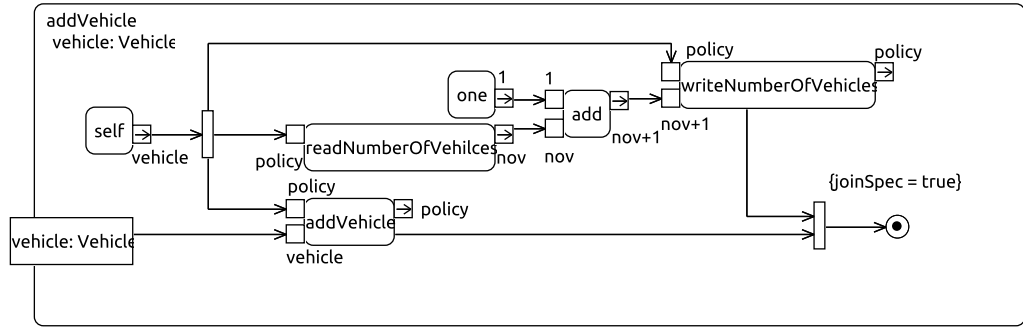
```

Figure 6 shows how the activity diagram benefits from a refactoring of the corresponding class diagram. Additionally a new activity diagram for the

`addVehicle` method instead of the old `addCar` and `addTruck` diagrams was created shown in 7. It is rather obvious that the complexity of both diagrams decreased on the visual level as well as on the level of execution.



**Fig. 6.** Calculate premium activity diagram with refactorings



**Fig. 7.** Add vehicle activity diagram resulting from the refactorings

## 4 Refactoring of fUML models

In the last section we presented an example model and its related activity diagrams. We also presented some of the possible refactorings such as *extract superclass* on the class diagram and their impact on the activity diagrams. This



section discusses how the models actually need to be transformed in order to realize the refactoring.

Every model has two kinds of syntaxes. The concrete syntax, which defines how the model looks like, and the abstract syntax, which defines the language elements and the grammar of the model. In order to refactor a model, we need to look at its abstract syntax and transform it according to the refactoring rules. In Figure 1 the refactorings are classified in order if the refactoring also implies changes in the meta-model representation. In the sections below we describe some examples that range from small to large impacts in the models.

#### 4.1 Rename refactorings

A refactoring takes several parameters to configure the parts of the model that change and how they change. Rename refactorings just need the new name as parameter. The respective classes, methods and attributes can then be adapted and the direct references be changed as the abstract syntax representation does not change. However in order to guarantee that the refactoring is possible, we have to ensure that there are no occurrences of the same type with the same name as the new name. In this work we did not take techniques such as method overloading into account. In our example there is no such case explicitly included.

#### 4.2 Extract superClass

The *extract superclass* refactoring takes two parameters, the new name of the extracted superclass as well as the list of classes that participate in the refactoring and which will receive the new class as their superclass.

In terms of the abstract syntax of these models, the two classes `Car` and `Truck` are instances of type *Class* of the meta-model a superclass named `Vehicle` would be an instance of type *Class* as well, and can be associated with these two classes through the recursive attribute *general* of the class *Class* to create the inheritance hierarchy. However the meta-model representation of the diagram does not change. In order to achieve a correct refactoring each class that participates in the refactoring must not already have a superclass. The model would also benefit from changing all possible references to the new class.

#### 4.3 Pull up attribute & pull up method

Both refactorings are quite similar which is why we describe the first and explain the specifics for the latter one afterwards. There are two cases of the *pull up attribute* refactoring. It can either be executed for an attribute that occurs in a single class or in multiple classes where the attributes should also be combined and pulled up. In the class diagram the attribute has to be pulled up and the occurrences of the attributes in the subclasses have to be deleted. The activity diagrams needs to change occurrences of the `StructuralFeatureValueAction` and use the attribute of the super class.

For the *pull up method* refactoring all `CallOperationAction` actions have to be adapted like described above. A precondition for both of these refactorings is that there does not already exist an attribute or a method with the same name in the target class and that they is not private.

#### 4.4 Encapsulate attribute

The *encapsulate attribute* refactoring is a rather complex one. At first setter and getter methods for the attribute have to be created. Afterwards the occurrences of the attribute have to be changed either to the setter for value assignments and to the getter for value usage. For the activity diagram the `WriteStructuralFeatureValueAction` has to be changed to a `CallOperationAction` which is a `BehavioralFeature` class instead of a `StructuralFeature` class in the meta-model. Finally the affected attribute has to be set to private. The precondition for this refactoring is that there are no methods with the same name in the classes that use the attribute.

### 5 Toolchain and implementation

For our toolchain we have relied on the moliz<sup>1</sup> repository, mainly for the ability to execute the fUML models in a virtual machine. The models are stored in XMI format and loaded with an EMF `ResourceSet`. The `ResourceSet` can be used to retrieve a `Resource` object through an URI, which is then used to access the different objects in the model (see figure 8). All elements contained in the model are instances of `Ecore` (the implementation of the MOF meta-meta-model) and they represent classes of the UML meta-model, which is implemented with `Ecore`.

```

1 resourceSet = new ResourceSetImpl();
2 resourceSet.getPackageRegistry().put(UMLPackage.eNS_URI,
3 UMLPackage.eINSTANCE);
4 resourceSet.getResourceFactoryRegistry()
5 .getExtensionToFactoryMap()
6 .put(UMLResource.FILE_EXTENSION,
7 UMLResource.Factory.INSTANCE);
8 File file = new File(umlModelFile);
9 URI uri = URI.createFileURI(file.getAbsolutePath());
10 resource = resourceSet.getResource(uri, true);

```

**Fig. 8.** Loading a UML model into a resource set

The extract superclass pre- and postconditions are shown in figure for single and multiple inheritance respectively. 9.

<sup>1</sup> <http://www.modelexecution.org>

```

1 Single Inheritance:
2 pre: self.superClass->isEmpty() and
3 self.owner.ownedElement
4 ->select(class|class.ocIsTypeOf(Class))
5 .oclAsType(Class).name
6 ->forAll(o | o<>newSuperClass.name)
7 post: self.superClass->size() = 1
8
9 Multiple Inheritance:
10 pre: self.owner.ownedElement
11 ->select(class|class.ocIsTypeOf(Class))
12 .oclAsType(Class).name
13 ->forAll(o | o<>newSuperClass.name)
14 post: self.general->contains(superClass)

```

**Fig. 9.** OCL pre and post conditions for extract superclass

These OCL constraints are checked by the refactoring for each class object that participates in the refactoring. An object to type OCL from the `org.eclipse.oc` package is used to create and evaluate these queries as can be seen in figure 10.

```

1 variable = ExpressionsFactory.eINSTANCE.createVariable();
2 variable.setName("newSuperClass");
3 variable.setType(UMLPackage.Literals.CLASSIFIER);
4 ocl.getEnvironment().addElement(variable.getName(),
5 variable, true);
6 query = helper.createQuery(
7 "self.general->includes(newSuperClass)");
8 eval = ocl.createQuery(query);
9 eval.getEvaluationEnvironment()
10 .add("newSuperClass", superClass);
11 if (!eval.check(clazz))
12 return false;

```

**Fig. 10.** OCL validation in Java

## 5.1 Model refactoring

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

## 5.2 GUI Integration

At the time of this writing our GUI integration is not finished, so we cannot describe it here. The idea is to use EMF.Refactor to add

## 6 Current Limitations

Our work currently has several limitations. So far we have designed a set of related class and activity diagrams, which are the basis for our refactorings. We have discussed the *extract superclass* refactoring including its pre- and post conditions and shown how it can be refactored with our toolchain. Until the final version of this paper we will add several more refactorings from the list shown in figure 1, such as the *extract class*, the *rename method*, the *pullup method*, the *pullup field* and *rename variable* refactorings.

We have also briefly mentioned EMF.Refactor and we plan to extend our toolchain such that our implemented refactorings can be invoked from the Eclipse GUI. At least we want to show that this is possible with our toolchain for at least one refactoring.

## 7 Related Work

In this section we give an overview on the related work. Refactoring in a general way with preconditions was described by Opdyke [15] in his master thesis and stated more precisely by Roberts [17] who also introduced postconditions for refactorings. Fowler [6] generated an extensive yet simple to understand catalog of refactorings for Java and Ruby which can be adapted to model refactorings.

Suny  et. al. [18] described how several refactorings can be applied to *UML* diagrams and introduced OCL as a possibility to specify pre- and postconditions. Gorp et al. [7] extends the discussion with the usage of *OCL* for additional analysis such as code smells of models.

Despite of the further discussion of model refactoring ([5], [3], [1]) most authors concentrate on static analysis and class diagram representations of models. Dynamic analysis of models by execution and debugging *fUML* models is discussed by Mayerhofer [9] and provides the basis for the approach discussed here. Mayerhofer et al. [10] furthermore introduce a runtime model and an implementation<sup>2</sup> that is capable to test the models and directly show impacts of refactorings.

Arendt and Taentzer [1] present a framework that is based on *Eclipse* and the *Eclipse Modelling Framework* which allows static model analysis and refactorings that are implemented in different languages (Java, OCL & Henshin) and can be directly extended in *Eclipse*.

---

<sup>2</sup> <http://www.modelexecution.org>

## 8 Conclusion

Refactoring models is a rather difficult task and it requires a concise knowledge of the involved technologies, with the list of involved technologies being rather long. For one a reasonably well understanding of fUML is required, in particular how class diagrams and activity diagrams are constructed and how they are related. It is not enough to simply be able to draw both class and activity diagrams, as one also needs a good understanding of the meta-meta-model (MOF) and the fUML and UML meta-model implementations in Ecore in order to understand and manipulate the model on the level of its abstract syntax. Finally an understanding of EMF (in particular Ecore and OCL) is required. While MOF and OCL are both modeling concepts and languages respectively, EMF contains implementations for them in Java with a complex API. Understanding these APIs of Ecore and OCL is required to perform the refactorings and was a prerequisite to build our toolchain.

With the toolchain in place another challenge was to identify the required steps to perform the actual refactoring work. In particular this meant to identify how the activity diagrams need to change if a change is made in the class diagram.

Most of our efforts in the development of this work were focused on gaining a comprehensive understanding of the technologies described above, to develop the tool chain and to draw the various diagrams presented in this paper and make them executable. We have presented a comprehensive set of diagrams as the basis for our refactoring work and demonstrated the feasibility of our tool chain. However a more comprehensive evaluation of the different refactorings including the specification of pre- and postconditions OCL in is still required and will be the focus of our ongoing efforts.

## 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. 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.
3. BAAR, T., AND MARKOVIC, S. A graphical approach to prove the semantic preservation of uml/ocl refactoring rules. In *Ershov Memorial Conference* (2006), pp. 70–83.
4. BECK, K. Embracing change with extreme programming. *IEEE Computer* 32, 10 (1999), 70–77.
5. CORREA, A. L., AND WERNER, C. M. L. Applying refactoring techniques to uml/ocl models. In *UML* (2004), pp. 173–187.
6. FOWLER, M. *Refactoring - Improving the Design of Existing Code*. AddisonWesley, July 1999.
7. 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.
8. MARKOVIC, S., AND BAAR, T. Refactoring ocl annotated uml class diagrams. *Software and System Modeling* 7, 1 (2008), 25–47.
9. MAYERHOFER, T. Testing and debugging uml models based on fuml. In *ICSE* (2012), pp. 1579–1582.
10. MAYERHOFER, T., LANGER, P., AND KAPPEL, G. A runtime model for fuml. In *Models@run.time* (2012), pp. 53–58.
11. 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.
12. OMG. *OMG Object Constraint Language*, 2.3.1 ed. OMG, <http://www.omg.org/spec/OCL/2.3.1>, 01 2011.
13. OMG. *OMG Unified Modeling Language*, 2.4.1 ed. OMG, <http://www.omg.org/spec/UML/2.4.1/>, 05 2011.
14. OMG. *Semantics of a Foundational Subset for Executable UML Models*, 1.1 ed. OMG, <http://www.omg.org/spec/FUML/1.1>, 08 2013.
15. OPDYKE, W. F. Refactoring object-oriented frameworks. Master’s thesis, University of Illinois, 1992.
16. RISING, L., AND JANOFF, N. S. The scrum software development process for small teams. *IEEE Software* 17, 4 (2000), 26–32.
17. ROBERTS, D. Practical analysis for refactoring. Master’s thesis, University of Illinois, 1999.
18. SUNYÉ, G., POLLET, D., TRAON, Y. L., AND JÉZÉQUEL, J.-M. Refactoring uml models. In *UML* (2001), pp. 134–148.