Computer Science and Information Systems 12(2):375–403 DOI: 10.2298/CSIS141025015H

Model-Driven Java Code Refactoring

Sohaib Hamioud[1], and Fadila Atil[2]

[1] Complex System Engineering Laboratory (LISCO), Department of Computer Science,
Badji Mokhtar University, POB 12,
23000 Annaba, Algeria
hamioud_sohaib@yahoo.com

[2] Complex System Engineering Laboratory (LISCO), Department of Computer Science,
Badji Mokhtar University, POB 12,
23000 Annaba, Algeria
atil_fadila@yahoo.fr

Abstract. Refactoring is an important technique for restructuring code to improve its design and increase programmer productivity and code reuse. Performing refactorings manually, however, is tedious, time consuming and error-prone. Thus, providing an automated support for them is necessary. Unfortunately even in our days, such automation is still not easily achieved and requires formal specifications of the refactoring process. Moreover, extensibility and tool development automation are factors that should be taken into consideration when designing and implementing automated refactorings. In this paper, we introduce a model-driven approach where refactoring features, such as code representation, analysis and transformation adopt models as first-class artifacts. We aim at exploring the value of model transformation and code generation when formalizing refactorings and developing tool support. The presented approach is applied to the refactoring of Java code using a prototypical implementation based on the Eclipse Modeling Framework, a language workbench, a Java metamodel and a set of OMG standards.

Keywords: code refactoring, metamodeling, model-driven engineering.

1. Introduction

Refactoring is the process of changing the internal structure of a software without changing its external behavior, typically aimed at making code more reusable, easier to maintain, easier to extend and easier to understand [3], [19]. Based on case studies [14], [36], refactoring is frequent and commonly practiced by programmers. Thus, automated refactorings were invented and are generally faster, more efficient and less error-prone than manual refactorings [22]. However, implementing a refactoring tool is still a challenging task.

Typically, most of the automated refactoring tool's code is devoted to the required components for building the tool. The most commonly used components are: (1) Lexer, parser and (sometimes) preprocessor for constructing a program presentation (generally an abstract syntax tree) which is used for both analysis and transformation. (2) Pretty printer for mapping back the performed changes to the source code. (3) Program database for indexing the program, and user interface for interacting with the tool. In fact, developers do not have to always construct refactoring tools from scratch. They sometimes reuse some existing components (e.g., parser, AST, syntax) from other tools (e.g., compiler frontends). However, reuse is not always possible and easy because existing components are originally constructed for different and specific concerns. This limitation was discussed in [20].

Yet another challenge that faces automation is handling language extensions. Java, for example, has been extended by new features over the last years (foreach loop, variable-length arguments, closures, etc.). Refactoring tools should be designed so they can be easily extended to support the manipulation of new language features. Furthermore, new refactorings (i.e., new preconditions, new postconditions and/or new transformations) should be easy to specify and easy to add. Within this context, a suitable approach is needed that can produce generic and extensible refactoring tools based on effective solutions.

This paper focuses on the application of model-driven techniques for implementing refactoring tools. Model Driven Engineering (MDE) adopts the use of models in software development for managing complexity, automating the process and raising the abstraction level. It is about creating, transforming, generating, interpreting and weaving models using modeling languages, tools, etc. The Object Management Group (OMG) introduced the Model Driven Architecture (MDA) initiative as an implementation of MDE [17]. It provides a suite of technologies and standards such as UML, OCL, XMI, CWM, and MOF.

The thesis of this paper is that rather than implementing the tool’s components (in particular lexer/parser and pretty printer) in a code-centric manner involving large amounts of handwritten code, one can automatically generate them from an explicit metamodel (abstract syntax) and textual syntax specification (concrete syntax) for the programming language, which not only reduces hand-coding, but also removes the burden of reusing existing components. This is possible with the use of modern language workbenches.

MDE increases the abstraction level where high level reuse (i.e., generative reuse) takes place, and this is achieved through modeling, transformations and code generation. Metamodels provide high expressiveness for describing languages and can be easily integrated and extended for reuse. Textual syntaxes and grammars (used to generate parsers and printers) can be reused as well along with metamodels [5], [6], [8].

While usual refactoring tools manipulate ASTs, here we manipulate models of the source code described according to the programming language metamodel. Hence, various existing modeling tools and standardized modeling and transformation languages can be used for this purpose. In this paper, we show how MDE can be applied to: (1) improve design and implementation of automated refactorings, (2) easily handle language extensions, and (3) make refactoring tools easy to extend and maintain.

The main components of a prototypical implementation are outlined. It contains eight useful refactorings: Rename (package, class, interface, method and field), Extract Local Variable, Move Method and Remove Class. We have chosen the Object Constraint Language (OCL) for static source code analysis [16], and the Query View Transformation (QVT) to apply refactoring (i.e., code transformation) [15]. The reminder of this paper is structured as following. The metamodel defining the abstract syntax of Java is presented in Section 2. Then Section 3 introduces an OCL-based approach for code analysis. The QVT-based code transformation is then integrated in Section 4. Section 5 outlines the several tools and techniques used to implement our experimental prototype.

In Section 6, the evaluation of the extensibility and the modification possibilities offered by our approach is introduced. Related work is presented in Section 7. Finally Section 8 concludes the paper and elaborates on future work.

2. Java Metamodel

Refactoring activities (analysis and transformation) must rely on a program model that conforms to an explicit structure describing the syntax and semantics of language entities. Our approach suggests the use of a metamodel that allows a model-based representation of the source code. Many Java metamodels exist and to choose the most suitable among them for use in a refactoring tool, there are some requirements that need to be met:

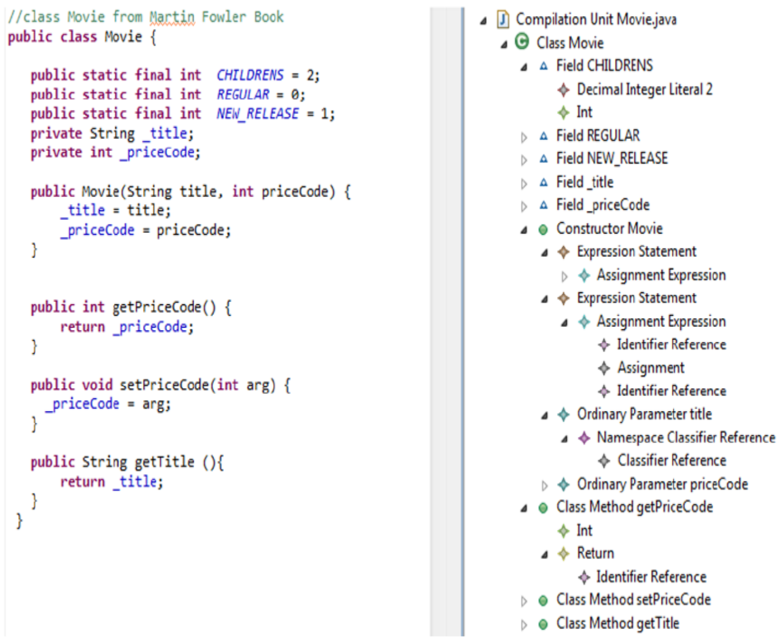
– *Granularity level*: a metamodel that provides coarse-grained elements like classes, methods and fields can be used to support the automation. However, this level of granularity limits the range of refactorings that can be implemented. Accordingly, lower granularity is required to cover, for example, statement level information, such as expressions and blocks;
– *Quality*: even if a fine granularity is required, the quality of the metamodel needs to be high so that developers (and users) can easily handle it, because, in our approach, they are supposed to use declarative languages to manipulate instances of the metamodel. In this declarative context, a high-quality metamodel must explicitly reflect a deep knowledge of the Java language in a clear and understandable way, which can be supported by subtyping (i.e., the use of subtyping relationship to express the identification and abstraction of common concepts used in different metaclasses that represent Java features), package structuring (i.e., the use of metapackages to group related metaclasses and metadatatypes), meaningful naming (i.e., the use of self-documenting names that clarify the intended use of each metamodel element. This also includes naming conventions) and high-quality referencing (i.e., definition of the most appropriate meta-associations between metaclasses). These techniques decrease complexity, promote understanding and support metamodel extensibility.
– *Completeness*: the need for a complete Java metamodel is apparent. Ideally, the layout of Java code must be preserved after both extraction (converting code into model) and generation (converting model into code). This is not possible if the metamodel does not cover the whole language. Moreover, details such as comments should not be omitted and lost;
– *Semantics modeling*: obviously, semantic analyses are essential for the successful implementation of refactoring. *Rename* (i.e., refactoring usable for renaming Java entities), for example, requires name binding analysis. Semantic information can be deduced statically or dynamically, but most of refactoring tools use static analysis. Fortunately, static semantics are sufficient to implement the most commonly used Java refactorings. As a result, a suitable Java metamodel needs to reflect static semantics;
– *Standardization*: with a standard-based approach like ours, the metamodel must be defined in a standard metamodeling language. This allows its integration with standard modeling tools.

In the literature, we found two Java metamodels that respond to the requirements: the SPOON metamodel [21] and the JaMoPP metamodel [7]. The later was chosen because, comparing to the former, it can easily be extended with new language features for Java. Moreover, we argue that the JaMoPP metamodel is better than the SPOON metamodel in term of quality. First, JaMoPP groups 233 metaclasses in 18 metapackages, while SPOON contains 70 metainterfaces and 70 metaclasses, which are divided into 4 metapackages. The higher number of metapackages used to organize JaMoPP elements promotes a better understanding. With regard to SPOON, three fundamental parts compose the metamodel: a structural part represented by the *structural* metapackage, a code part represented by the *code* metapackage and a reference part represented by the *reference* metapackage. JaMoPP goes beyond that by splitting these parts into more accurate groups. For example, the code metapackage contains the executable Java code found in method bodies. JaMoPP models this part using several metapackages including *statements*, *expressions*, *operators*, *arrays* and *variables*. Yet another fact that supports our argument is that JaMoPP exploits subtyping better than SPOON. While JaMoPP defines 18 abstract metaclasses, the SPOON metamodel does not contain any. Let's take as example the concept of Class. In JaMoPP, a *Class* is a *ConcreteClassifier* which is a *Classifier*. A *Classifier* is a *Type* and a *ReferenceableElement*. A type is a *Commentable*, and a referenceable element is a *NamedElement*. As one can notice, JaMoPP pushes common concepts (e.g., meta-attributes *name* and *comments*) into abstract metaclasses (e.g., *NamedElement* and *Commentable*) which reduces redundancy and increases understandability.

JaMoPP's metamodel is defined in the metamodeling language ECORE [32] which is a widely used implementation of the OMG standard Essential MOF (EMOF). To support semantic analyses, JaMoPP represents static semantics through cross-references between model elements (type information, class hierarchy, method calls, name bind-ings, etc.). It covers the whole Java language including annotations and generics (by means of metaclasses contained in *annotations* and *generics* metapackages respect-ively). JaMoPP brings not only a complete Java metamodel with a fine granularity but also a model-level representation of Java code.

Fig. 1 illustrates a model-level representation (*right*) of a Java class (*left*) conforming to JaMoPP's Java metamodel. It is the EMFText Java editor outline view of the Movie class in the form of a tree structure. For the sake of simplicity, we removed keyword layout information. Each model element in the tree has a corresponding element in the textual representation. Only some important elements are shown, i.e., instances of the metamodel's concrete metaclasses, references (meta-association ends) and required fields (multiplicity-1 meta-attributes). Comments do not appear in the view, but rather are implicitly stored in the String-typed field *comments* inherited from the abstract metaclass *Commentable*. They are associated with the following, preceding or surround-ing model element (e.g., the line comment in the Movie class is associated with the *Public* modifier of the *Class*).

Relations between models and metamodels are explained by the OMG' MDA standard (Fig. 2) where metamodels are described using the OMG' Meta Object Facility (MOF) metametamodel [18]. At the metamodel level, JaMoPP is adopted to define the concepts of the Java programming language. All models conforming to JaMoPP are represented as EMF models [32]. They provide a model-level representation of Java programs which are on the instance level.



```
//class Movie from Martin Fowler Book
public class Movie {

   public static final int  CHILDRENS = 2;
   public static final int  REGULAR = 0;
   public static final int  NEW_RELEASE = 1;
   private String _title;
   private int _priceCode;

   public Movie(String title, int priceCode) {
       _title = title;
       _priceCode = priceCode;
   }


   public int getPriceCode() {
       return _priceCode;
   }

   public void setPriceCode(int arg) {
     _priceCode = arg;
   }

   public String getTitle (){
       return _title;
   }
 }
```

```
Compilation Unit Movie.java
  Class Movie
    Field CHILDRENS
      Decimal Integer Literal 2
      Int
    Field REGULAR
    Field NEW_RELEASE
    Field _title
    Field _priceCode
    Constructor Movie
      Expression Statement
        Assignment Expression
      Expression Statement
        Assignment Expression
          Identifier Reference
          Assignment
          Identifier Reference
      Ordinary Parameter title
        Namespace Classifier Reference
          Classifier Reference
      Ordinary Parameter priceCode
    Class Method getPriceCode
      Int
      Return
        Identifier Reference
    Class Method setPriceCode
    Class Method getTitle
```

Fig. 1. Model-based representation of Java class Movie from Fowler's Book [3]

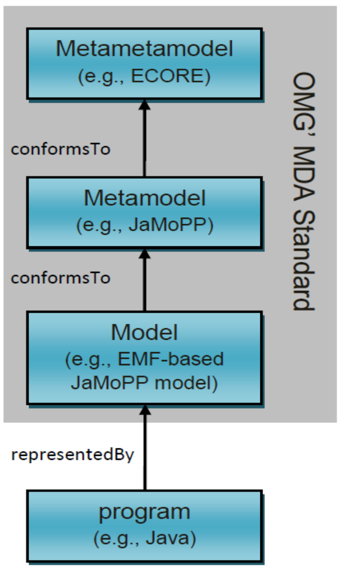


Fig. 2. MDA four-layer architecture

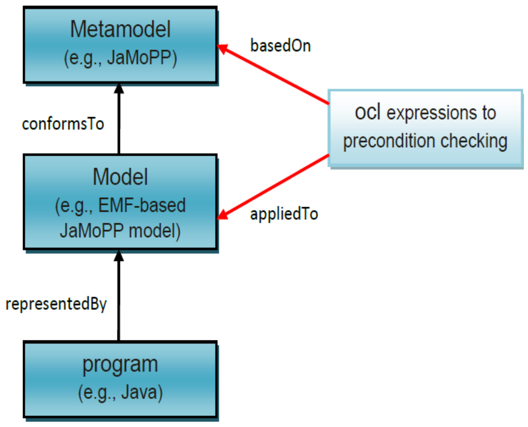


Fig. 3. Model-driven precondition checking

3. OCL-based Code Analysis

Typically, automated refactoring tools analyze source code, then transform it. Code analysis can be done in two ways: static and dynamic. Both methodologies complement each other in a number of dimensions. Static analysis is a program-centric process, performed on a program model (i.e., information abstraction). In contrast, dynamic analysis is an input-centric process since it depends on the test suite used to evaluate, trace and traverse the program during runtime. Within a refactoring context, tools generally use static analysis, especially for statically typed languages like Java. This can be explained by the fact that dynamic analysis is relatively slow, as good as the designed test suite and generates large amount of data. Another thing that promotes the use of static analysis is that the most implemented refactorings in current tools require neither dynamic semantics nor complicated static analysis.

As said before, refactorings must preserve the external behavior of the program. Generally, developers ensure behavior preservation in their tools by enforcing a set of preconditions that the program must satisfy so that transformation can proceed safely, which is much harder to implement in tools than the code transformations. We adopt an OCL-based code analysis for checking preconditions. Our approach for static analysis is tied to the code representation, in this case models, since we used a constraint and query language that must be applied to models.

OCL is an evident choice to query source code [16]. Being a standard language, OCL is powerful enough to express the necessary queries and conditions on Java models for behavior-preserving refactorings. As shown in Fig. 3, OCL is tightened by the target language metamodel due to the fact that declarative OCL expressions must be defined in the context of a metamodel element.

Here, preconditions are given in terms of analysis operations (primitive and derived) similar to the ones given in [22]. The specification of these operations is based entirely on OCL (version 2.3.1). An example of a primitive OCL analysis operation can be found in Listing 1.

```
Context
  java::containers::Package::
  hasClassifier(classifierName:String):Boolean
body:
  self.getClassifiersInSamePackage()->exists(c|c.name=
classifierName)
```

Listing 1. OCL primitive analysis operation hasClassifier(classifierName)

The OCL expression defining the operation is specified against the JaMoPP meta-model in the context of *Package*, which is a *JavaRoot*. It checks the existence of the classifier named *classifierName* in the package. A classifier can be a *Class*, *Interface*, *Enumeration* or *Annotation*. The primitive operation *hasMember*(*memberName*) (Listing 2) returns true if a member with a given name exists in the target *MemberContainer*.

```
Context

java::members::MemberContainer::hasMember(memberName:Stri
ng):Boolean
body:
  self.members->exists(m|m.name= memberName)
```

Listing 2. OCL primitive analysis operation hasMember(memberName)

```
Context

java::classifiers::Class::superClass():java::classifiers:
:Class
body:
  self.getSuperClass()
```

Listing 3. OCL primitive analysis operation superClass()

Listing 3 contains the primitive operation *superClass()* which returns the immediate superclass of *self* (i.e., a given class). Other primitive analysis operations are shown in Listings 4 and 5. *isReferenced()* is defined in the context of *Class* and allows to verify whether or not a given class is referenced. This operation leads to an overly strong precondition because it does not allow the remove refactoring if the class to be removed is referenced internally or imported but never instantiated. This restriction can be relaxed so that a class is removed only if it is not referenced externally (see section 6.3). *isEmpty()* returns true if the class has no methods and no fields.

```
Context
  java::classifiers::Class::isReferenced():Boolean
body:
  java::types::NamespaceClassifierReference.
  allInstances()->
  exists(ncr|
         ncr.classifierReferences->
         exists(cr|

cr.target.oclIsTypeOf(java::classifiers::Class) and

cr.target.oclAsType(java::classifiers::Class)= self)
         ) or
  java::references::IdentifierReference.allInstances()->
  exists(ir|
          ir.target.oclIsKindOf(java::members::Member)and
          ir.target.oclAsType(java::members::Member).
          getContainingConcreteClassifier().oclIsTypeOf
          (java::classifiers::Class) and
          ir.target.getContainingConcreteClassifier().
          oclAsType(java::classifiers::Class)= self
        ) or
  java::references::MethodCall.allInstances()->
  exists(mc|

mc.target.oclIsTypeOf(java::members::ClassMethod) and

mc.target.oclAsType(java::members::ClassMethod).
          getContainingConcreteClassifier().
          oclAsType(java::classifiers::Class)= self
        ) or
  java::imports::ClassifierImport.allInstances()->
  exists(ci|

ci.classifier.oclIsTypeOf(java::classifiers::Class) and

ci.classifier.oclAsType(java::classifiers::Class)= self)
```
Listing 4. OCL primitive analysis operation isReferenced()

In our approach, some of the analysis operations used to describe preconditions are derived from the primitive operations. For example, the analysis operation *subClasses()* (which returns the set of all immediate subclasses of a given class) is specified based on the primitive operation *superClass()* as shown in Listing 6.

```
Context
  java::classifiers::Class::isEmpty():Boolean
body:
  let members:Set(java::members::Member)= self.members in
  members->isEmpty() or
  members->select(m|
                  m.oclIsTypeOf(java::members::Field)
                  or

m.oclIsTypeOf(java::members::ClassMethod)
                )->isEmpty()
```

Listing 5. OCL primitive analysis operation isEmpty()

```
Context java::classifiers::Class::subClasses():
  set(java::classifiers::Class)
body:
  java::classifiers::Class.allInstances()->select(c|

c.superClass()= self
                                               )
```

Listing 6. OCL derived analysis operation subClasses()

```
Context

java::classifiers::Class::renameClass(newName:String):Boo
lean                                      body:
  not self.getContainingPackage().hasClassifier(newName)
  and
  not
self.oclAsType(java::members::MemberContainer).hasMember(
newName)
```

Listing 7. Precondition specification of the renameClass refactoring in terms of OCL analysis operations

For instance, to rename a class, a precondition must be considered and checked before the renaming, to avoid any unexpected program behavior. This precondition consists of three subconditions: (1) the given name is valid (not null, begins with a letter or underscore and contains only letters, digits, and underscores), (2) there exists no classifier (i.e., class, interface, enumeration or annotation) with a name identical to the new name in the containing package and (3) the new name is distinct from any other member names declared in the class. The validity verification of the new name is done using Java since it is more related to the user interface component than to the meta-model; the other two conditions are represented by two primitive operations. The precondition specification of the *rename class* refactoring is shown in Listing 7.

As yet another example, consider the preconditions of the *removeclass* refactoring (Listing 8). A class to be removed must be unreferenced, which is guaranteed using the primitive operation isReferenced(). This class should either have no subclasses or have subclasses but have no methods or fields. Derived operation subClasses() and primitive operation isEmpty() allow to guarantee that. Because of space limitations we omit more examples for OCL analysis operations.

```
Context
  java::classifiers::Class::removeClass():Boolean
body:
  not self.isReferenced()and
  (self.subClasses()->isEmpty()or self.isEmpty())
```

Listing 8. Precondition specification of the removeClass refactoring in terms of OCL analysis operations

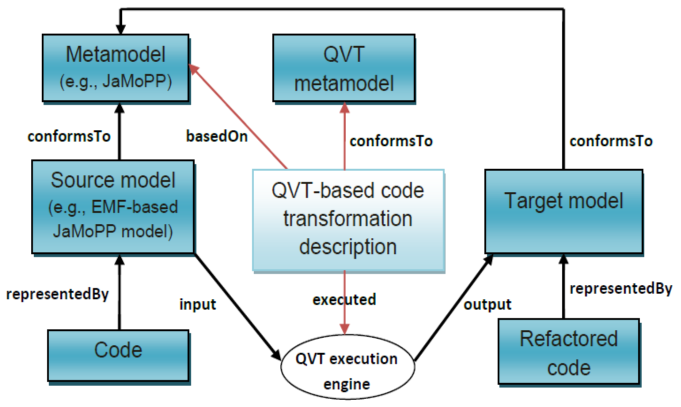


Fig. 4. Model-driven code transformation

4. Model-based Code Transformation

Relying on the results of precondition checking, a code transformation must be applied if allowed. Whereas a large number of refactoring tools use the AST to transform the source code by an AST rewriter, we execute transformations on the model built from original code using a standardized transformation language. The transformation makes changes to code by adding, moving, removing or modifying elements in the model. As depicted in Fig. 4, this kind of transformation is *endogenous* because the source and the target language metamodel are the same. Extraction and generation steps must be carried out before and after applying model transformations, respectively. The output of the extraction step and the input of the generation step are EMF models conforming to JaMoPP. Its elements are used in the QVT transformation definition that conforms to the QVT metamodel. This definition will be executed by the QVT execution engine which, according to its transformation rules, reads and transforms a source model representing a source Java code to a target model representing the desired refactored Java code.

Model-based Java code representation enables to apply generic modeling tools like QVT [15]. Specifically, Operational QVT (QVTO) is the transformation engine used in the implementation of automated refactorings. With the *Java BlackBoxing* mechanism, QVT opens up the possibility for calling external Java libraries. Moreover, the QVT specification integrates the OCL.

```
import m2m.qvt.oml.RefactorLib;
modeltype java uses "http://www.emftext.org/java";
transformation createClass(out javaModel:java);
... ... ... ...
main()
{  map createCompilationUnit();
}
mapping
createCompilationUnit():java::containers::CompilationUnit
{
   var namespaces: Sequence(String)= getNameSpaces();
   result.namespaces:= nameSpaces;
   result.classifiers+= map createClass();
}
mapping createClass():java::classifiers::Class
{
  var modifiers:= getModifiers();
  result.name:= getClassName();
  Sequence{1.. modifiers->size()}->forEach(i|
    let modifier:String= modifiers->asSequence()->at(i)
    in
```

```
      if (modifier= 'public') then
         result.annotationsAndModifiers+=
         object java::modifiers::Public{}
      else if (modifier= 'abstract') then
         result.annotationsAndModifiers+=
         object java::modifiers::Abstract {}
      else result.annotationsAndModifiers+=
         object java::modifiers::Final {}
      endif
      endif
    )
  }
```

Listing 9. QVT transformation for the application of the createClass refactoring

Like in [19], we adopt a decomposition approach in which primitive refactorings can be composed to form more complex refactorings (composite refactorings). A primitive refactoring creates, deletes, renames, modifies or copies entities (e.g., packages, classes, interfaces, attributes, methods, parameters and variables). A composite refactoring is a combination of primitive or composite refactorings. The transformation is configured according to the selected refactoring via the user interface, and is invoked with Java. For example, Listing 9 shows the QVTO transformation code of the primitive refactoring *createClass*. The first three lines represent the code which imports the blackboxing library, and defines the JaMoPP metamodel and the transformation header.
The main function of the transformation creates a compilation unit (i.e., a Java file) in the given package. An empty class with a specified name and access modifiers is created within the compilation unit. It has no members, super or subclasses. Java is used to identify which package the user selected and what arguments were given (e.g., class name, package name and modifiers). As shown in the example above, three operations are needed from the imported blackboxing library:

- *getNamespaces()*: returns the package namespaces as a sorted list of Strings. The transformation sets the *namespaces* attribute of *CompilationUnit* to the package name;
- *getClassName()*: returns the name of the new class. The transformation sets the *name* attribute of *Class* to the specified name. This operation provides a default name if one is not given;
- *getModifiers()*: returns the modifiers of the class as a sorted list of Strings. The transformation sets the *annotationsAndModifiers* attribute of *Class* to the given set of modifiers. If no modifier is selected, the operation will return an empty list. Only *public, abstract* and *fina*l access modifiers are allowed when creating a class using the user interface.

5. Implementation

According to [13], [14], *Rename*, *Extract Local Variable* and *Move* are, in this order, the most used automated refactorings. To apply them following the presented approach, we used a set of standards and modeling tools. The first task is modeling, since the language metamodel (abstract syntax) must be defined, along with the concrete syntax, to generate, using a language workbench, the infrastructure underlying the automation process (i.e., a parser and a printer).

Both analysis and transformation can rely on two distinct metamodels, because our approach treats both of them separately. It depends on the language, the refactorings to be implemented and developers' choices. In scenarios where a simplified language metamodel, which is sufficient to cover the required analyses, exists or can easily be defined, one can use it for analysis rather than using the complete metamodel, which results in simplified OCL expressions. However, for general-purpose programming languages like Java, two metamodels require an extended infrastructure which can lead to a sizeable tool. For that reason, we used a single metamodel for both analysis and transformation.

5.1. Tools

A number of tools have been used to support our approach as depicted in Fig. 5. The Eclipse platform and its open universal IDE (Integrated Development Environment) present a tool integration framework where different tools can be integrated as plugins to add functionality [33]. Since modeling is a fundamental part of our implementation, we used EMF (Eclipse Modeling Framework) which is a modeling framework for building tools and other applications based on a structured data model [32]. EMF models are the foundation for fine-grained data integration in Eclipse and, for OCL-based analysis, they can be analyzed using the OCL implementation provided by the Eclipse OCL project [34]. Another part of the Eclipse modeling project is the Model to Model Transformation (MMT), its sub-project QVT Operational [35] is our choice for implementing the refactorings.

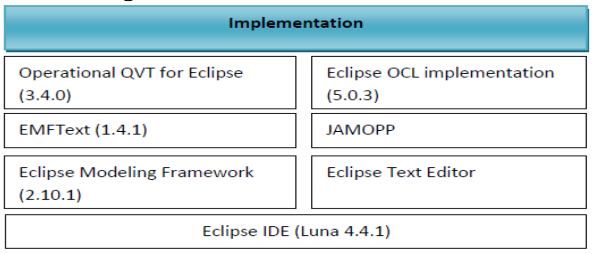

Implementation
Operational QVT for Eclipse (3.4.0)
Eclipse OCL implementation (5.0.3)
EMFText (1.4.1)
JAMOPP
Eclipse Modeling Framework (2.10.1)
Eclipse Text Editor
Eclipse IDE (Luna 4.4.1)

Fig. 5. Tools underlying the implementation

There are modern tools for defining textual languages, whether general purpose (GPLs) or domain specific languages (DSLs). EMFText [6] is a good example (Fig. 6); it bridges the gap between abstract and concrete syntax and provides the possibility of deriving a generic syntax specification automatically from the former. This specification will be refined using EMFText facilities to define the concrete syntax and then generate the tooling for the language (e.g., parser, printer, textual editor supported by syntax highlighting, code completion, quick fixes, reference resolution and refactoring). EMFText is tightly integrated with EMF which provides facilities to handle models such as resource management (e.g., for loading and saving models). Ecore is the commonly used metamodeling language of EMF. In addition to the fact that Ecore is a standard, it allows the language metamodel to be processed by existing Ecore-based modeling tools.

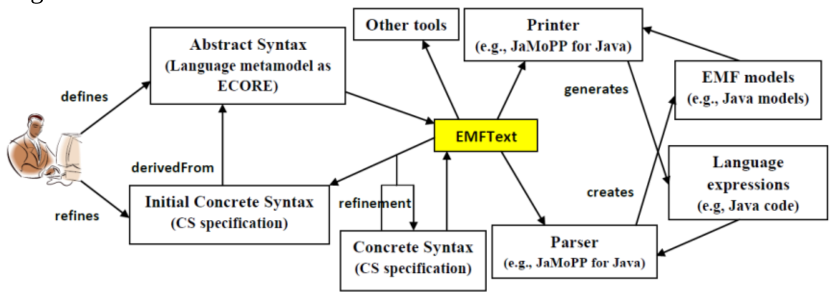

Abstract Syntax (Language metamodel as ECORE)
Other tools
Printer (e.g., JaMoPP for Java)
EMF models (e.g., Java models)
defines
generates
EMFText
derivedFrom
refines
Initial Concrete Syntax (CS specification)
refinement
creates
Language expressions (e.g, Java code)
Concrete Syntax (CS specification)
Parser (e.g., JaMoPP for Java)

Fig. 6. Overview of EMFText

With the help of EMFText, JaMoPP has been developed for the Java programming language to build models from their respective code and vice versa [5]. From a complete Ecore metamodel for Java, an initial text syntax specification, which necessitates further modifications, was generated by a mechanism that conforms to the HUTN standard. EMFText has a language called ConcreteSyntax (CS) for specifying text syntax. This language is based on the Extended Backus-Naur Form (EBNF). Thanks to the obtained CS specification of the Java syntax, JaMoPP provides a parser and a printer, which play a crucial role in refactoring engines.

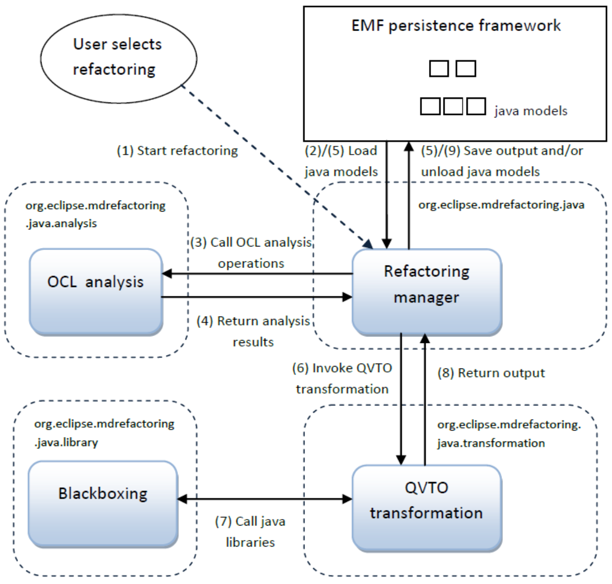

User selects refactoring
EMF persistence framework
java models
(1) Start refactoring
(2)/(5) Load java models
(5)/(9) Save output and/or unload java models
org.eclipse.mdrefactoring .java.analysis
org.eclipse.mdrefactoring.java
(3) Call OCL analysis operations
OCL analysis
Refactoring manager
(4) Return analysis results
(6) Invoke QVTO transformation
(8) Return output
org.eclipse.mdrefactoring .java.library
org.eclipse.mdrefactoring. java.transformation
Blackboxing
QVTO transformation
(7) Call java libraries

Fig. 7. Workflow of the refactoring process

5.2. Workflow

Rename (package, class, interface, method and field), *Extract Local Variable*, *Move Method* and *Remove Class* are the automated refactorings we selected for implementation as a plugin to the eclipse environment. Fig. 7 shows the workflow of the refactoring process and the plugin components involved.

Once the user selects a refactoring, the *Refactoring manager* looks for Java models needed for the analysis phase, and calls the respective OCL analysis operations. EMF is indispensable for the implementation; it provides a powerful framework for model persistence. Java files are represented by *resources*. A resource is the basic unit of persistence in EMF. We determined the persistence form (i.e., JaMoPP resource factory) and registered it with the EMF's resource factory registry interface. Specifically, the *ResourceManager* class of the Refactoring manager component is responsible for loading, saving and unloading Java models resources. EMF persistence framework includes an interface called *Resourceset* which is used to manage references within one resource as well as between different resources (cross-references). The *OCL analysis* component (org.eclipse.mdrefactoring.java.analysis) evaluates the OCL operations that specify the precondition of the selected refactoring on the corresponding Java models. It then passes the analysis results to the Refactoring manager, which sends the models to the QVTO transformation engine only if the precondition is satisfied. The transformation code (contained in a resource) is invoked and executed automatically by the Refactoring manager. The output is saved and the target models are converted into well-formatted Java code.

The Refactoring manager plays an important role in the refactoring process, because it initiates, based on the user interface, both the analysis and the transformation**,** and handles the EMF resource management. Since there are composite refactorings that may take place, there may be several iterations of the sequence of steps from 2 to 9 (see Fig. 7).

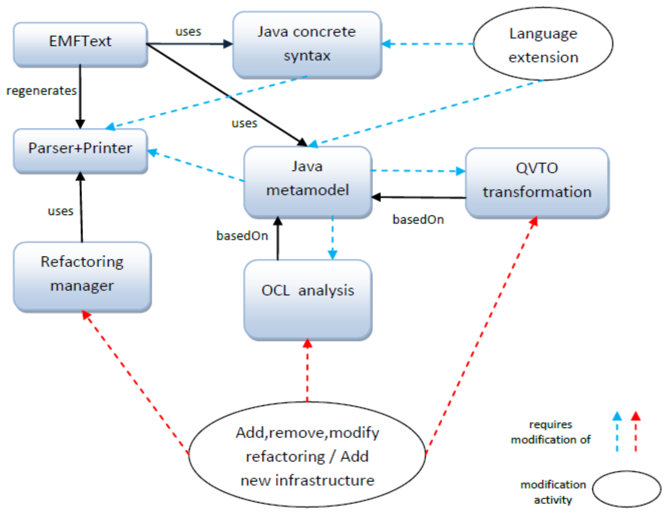
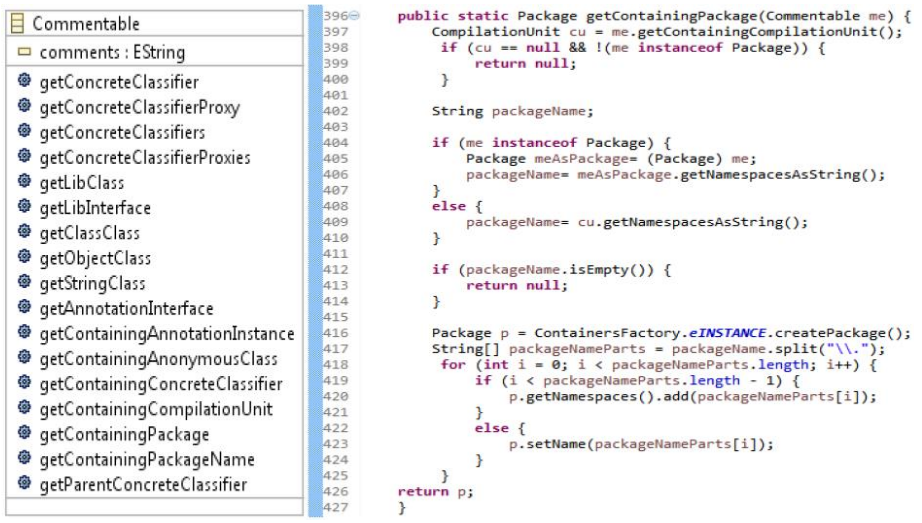


Fig. 8. Tool extension and customization

6. Evaluation

In the evaluation phase, the following question is addressed: to what extent the tools implemented using our approach are flexible to accommodate modifications? To answer this question, we consider several extension scenarios of our prototype where different parts of the tool are concerned. The experiment proved that the automated support implemented following our approach is easily extensible and highly customizable. This is explained by the extensibility of these artifacts: the language metamodel, the language syntax, the OCL analysis operations and the QVTO transformations. The main modification activities and the concerned artifacts are summarized in Fig. 8. A language extension can be done by extending the metamodel and the text syntax, and regenerating the refactoring engine (i.e., the parser and the printer). Typically, changing the metamodel also requires changing some of the analysis operations and the transformation rules. A new refactoring can be implemented by giving its OCL precondition and the appropriate QVT transformation code. The opposite can be done to remove an existing refactoring. Moreover, considering its important role in our implementation, the Refactoring manager component should be maintained when the tool evolves. Another possibility would be the extension of the tool infrastructure to support new languages (DSLs or GPLs), which is equivalent to adding new parsers and printers (i.e., new metamodels), new preconditions and new transformations. Generally, developers should determine what modifications need to be applied on which artifacts.



```
public static Package getContainingPackage(Commentable me) {
    CompilationUnit cu = me.getContainingCompilationUnit();
     if (cu == null && !(me instanceof Package)) {
         return null;
     }

    String packageName;

    if (me instanceof Package) {
        Package meAsPackage= (Package) me;
        packageName= meAsPackage.getNamespacesAsString();
    }
    else {
        packageName= cu.getNamespacesAsString();
    }

    if (packageName.isEmpty()) {
        return null;
    }

    Package p = ContainersFactory.eINSTANCE.createPackage();
    String[] packageNameParts = packageName.split("\\.");
     for (int i = 0; i < packageNameParts.length; i++) {
         if (i < packageNameParts.length - 1) {
             p.getNamespaces().add(packageNameParts[i]);
         }
         else {
             p.setName(packageNameParts[i]);
         }
     }
return p;
}
```

Fig. 9. Extending metaclass Commentable with the EOperation getContainingPackage()

6.1. Extending the Language Metamodel

The language metamodel can be extended to support new language features or to be enriched with an additional useful constructs. In this section, we present an example of the latter case (i.e., adding useful constructs), where only the metamodel is extended without affecting the language (i.e., the syntax). We enriched the original JaMoPP metamodel with the operation *getContainingPackage()* (used in the rename class precondition specification of Listing 7) which is defined in the metaclass *java::commons:: Commentable*. This operation returns the containing package of type *java::containers:: Package*. The main reason behind the definition of this operation is to provide the possibility to create an explicit representation of the package that contains the method caller.

One might argue that getting the containing package object is a redundancy because, instead of accessing the package namespaces and the contained compilation units from this object, it is possible to use common operations created by JaMoPP for this purpose, such as *getContainingPackageName()* which returns the name of the containing package, *getContainingCompilationUnit()* which returns the containing compilation unit and *getClassifiersInSamePackage()* which returns all classifiers contained in a given package. Technically, this is true; however, it is conceptually wrong to define OCL operations that inspect Java code in a package (e.g., primitive operation *hasClassifier(classifierName)* of Listing 1) in the context of a CompilationUnit or a Classifier, since Package is the parent container of these entities.

Fig. 9 shows the modified metaclass Commentable (left) and the behavior specification of the getContainingPackage() method (right) of the corresponding Java class. Taking performance into consideration, package objects created by this method are "lightweight" as they do not contain any compilation unit. Once the package is obtained, JaMoPP's common operations can be used to fully exploit it.

Although this example is simple and concerns only the metamodel, it is obvious how simple and straightforward the extension of the metamodel is. Accordingly, developers can easily adjust it to fit their needs in the way we did it. Next, we will show a more complex example where Java is extended with new features.

6.2. Extending the Language

Programming languages evolve over time by adding new features. Java is a prominent example, where eight versions were released since the initial introduction in 1996. Also, extending a general purpose language like Java by embedding within it a domain specific language to accomplish specific tasks is often done in practice. Refactoring tools must handle these extensions in an easy and efficient manner. The problem is that parsing a composite program written in Java and the embedded DSL constructs is challenging. Seeing that parsing is a very important step, not only in refactoring, but also in syntactic and semantic analysis activities (e.g., compilation), the implemented tooling that supports the embedded language has to handle the composite grammar efficiently and keep the existing tools (e.g., refactoring tool) aware of the extension. This is generally not the case.

To shed light on this issue, we take as example the Tom language [1], [12]. Tom is a DSL designed to extend GPLs with constructs to manipulate tree structures and XML documents. It provides a powerful pattern matching and term rewriting features. Tom distribution for Java includes a plugin for the eclipse IDE, which provides a textual editor supported by syntax highlighting and code completion. Unfortunately, besides rename and move (resource and package), other refactorings are not supported. This is expected, especially if we know that Tom and Java can be unboundedly nested which makes parsing even more difficult. Consequently, reusing the implemented JDT refactorings is not possible without an immense work. Like refactoring, syntax error detection is also hindered by the complex nature of the two languages mixture. In Tom, Java code is considered as a sequence of characters. The Tom compiler traverses the program and generates corresponding Java program. In other words, two compilation steps are required to run a Tom program. Accordingly, Java syntax errors are not detected until the compilation of the generated code. Even with the propagation of syntax errors from the generated code to Tom code, refactoring still challenging considering the representation of Java code in Tom. Furthermore, refactoring the generated code using the JDT refactoring tool does not solve the problem, because propagating changes in the other direction (i.e., from Java to Tom) is not supported.

The code of Listing 10 is a Tom program that defines the algebraic data-type Peano to represent Peano integers, and builds the integers 0=zero and 1=suc(zero). The %gom {…} construct defines the sort *Nat* and its operators *zero()*, *suc(Nat)* and *plus(Nat,Nat)*, which are used as constructors to build the data-structure. These operators represent arities and possess zero or more arguments (slots). Each argument has a name and a type (sort). The compilation of this code results in the generation of a corresponding Java code. From the data-type definition, an API is generated in a default Java package named *types* and its namespaces are the name of the Tom file (ignored if the data-type is defined in a separated Gom file) followed by the module name in lower case letters (e.g., the code generated by the %gom{..} in the example can be found following this path: main\peano\types, where *main* is the Tom file name lowercased). Also, a Java compilation unit that contains the same Java code as the Tom file is generated, where the only difference is that the Tom features are replaced by Java equivalents. Sorts and operators are translated into Java classes in the package *types*. This explains the inclusion of the import statement since these classes are used to manipulate the data-type. The back-quote (`) construct is translated into method calls and is used in the example to initialize the variables *z* and *one* of type Nat.

```
import main.peano.types.*;
  public class Main {
  %gom {
    module Peano
    abstract syntax
    Nat = zero()
        | suc(pred:Nat)
        | plus(x1:Nat, x2:Nat)
  }
  public final static void main(String[] args) {
    Nat z = `zero();
    Nat one = `suc(z);
    System.out.println(z);
```

```
      System.out.println(one);
    }
  }
```

Listing 10. Tom program of the Tom file Main

Using the available refactorings (i.e., rename and move) is sufficient to know the limits of a JDT-based refactoring. For instance, renaming the Tom file Main to Main2 results in the generation of new Java implementation in new packages (main2/peano/ types) and the generation of a new Java compilation unit named *Main2* with syntax errors. This is explained by the fact that renaming a Tom file is different from the renaming of a Java file. The rename refactoring of a Java compilation unit implies the renaming of the contained class and the updating of all references to this class, which is not the case when this refactoring is applied on a Tom file. First, the refactoring changes only the name of the Tom file and does not change the contained class name, because of the parsing approach adopted by Tom. Second, the transformation must rename the package and the corresponding Java compilation unit (if exist) before renaming the actual Tom file to avoid the creation of undesirable new packages and duplicated Java files. Obviously, the JDT refactorings cannot be used to refactor Tom programs without a thorough revision of the provided API.

Our approach defines the language concepts at a high level of abstraction making this kind of extension easier. To solve the problems explained above, the first thing to do is to evade the two-phase compiling approach by creating a parser capable of recognizing the two languages constructs. In our approach, this is equivalent to the creation of a metamodel that defines the two languages concepts and describes how they are nested. An excerpt of such a metamodel is illustrated in Fig. 10. JaMoPP is used as a basis to perform the Java extension by integrating the Tom concepts. The extended metamodel defines the relationships between Tom and Java. For example, a *GomModule* is defined as a *ConcreteClassifier* which allows the recognition of gom%{..} constructs as class members and referenceable elements. The definition of a *DataType* as a *Concrete Classifier* has the advantage of treating Tom data-types just as any other Java types. Operators are represented by the metaclass *GomOperator* which is integrated in JaMoPP as a *ReferenceableElement* (since operators are referenced by other elements) and a *Parametrizable* (since an operator can have parameters). The back-quote construct used in the example (for building the data-structure) is represented by the metaclass *BackquoteTermBuilder* which is an *ElementReference* (since it references a constructor) and an *Argumentable* (to represent the constructor arguments). All Tom metaclasses are defined in the metapackage *tom*. Listing 11 shows the model (serialized as an .xmi file) describing the Tom program of Listing 10, which conforms to the extended metamodel.

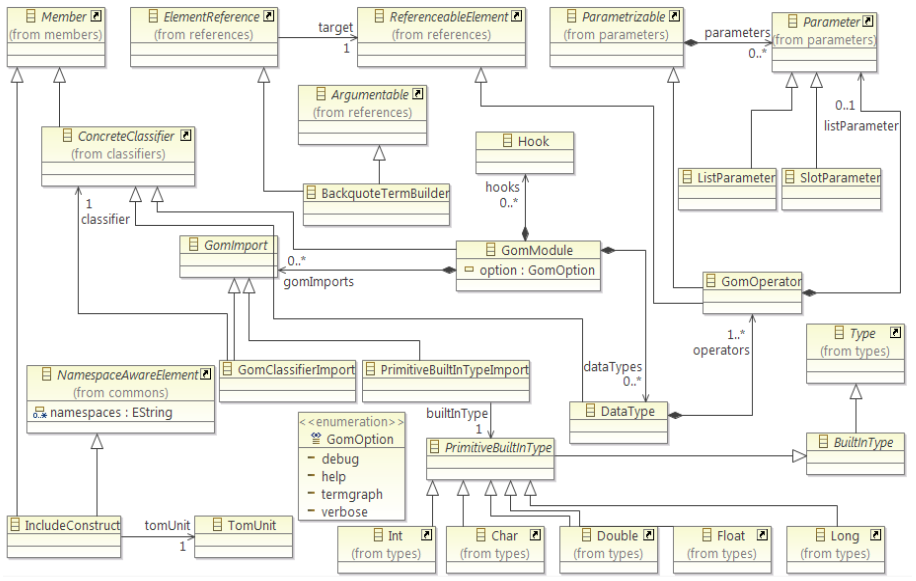


Fig. 10. Excerpt of the Tom metamodel

The concrete syntax has to be extended as well to introduce the Tom syntax. Thanks to the importing mechanism offered by EMFText, Java rules are reused and only the rules defining the Tom syntax must be given. Resolving references (especially cross-resource references) is the most difficult task of this experiment. EMFText generates a set of resolvers but, in our case, they cannot be used to resolve references without adjustment, since some resolving rules are Java-specific and, in addition to the fact that the JaMoPP resolvers are originally adjusted, some disambiguation procedures must be hand-coded to resolve ambiguities resulted from the unbounded nesting of Tom and Java. For example, the generated parser, without adjustment, cannot distinguish if a non-primitive type is a class, a module or a data-type. Consequently, the type String of argument *args* of the method *main* is recognized as a Tom data-type instead of a Java class. Besides resolving challenges, the language extension is straightforward.

```
<containers:CompilationUnit>
 ...
 <classifiers xsi:type="classifiers:Class" name="Main">
  <members xsi:type="tom:GomModule" name="Peano">
   <dataTypes name="Nat">
    <operators name="zero"/>
    <operators name="suc">
     <parameters xsi:type="tom:SlotParameter"name="pred">
      <typeReference
xsi:type="types:NamespaceClassifierReference">
```

```
       <classifierReferences>
        <target xsi:type="tom:DataType" href=

"//@classifiers.0/@members.0/@dataTypes.0"/>
       </classifierReferences>
      </typeReference>
     </parameters>
    </operators>
    <operators name="plus">
       ...
    </operators>
   </dataTypes>
  </members>
  <members xsi:type="members:ClassMethod" name="main">
     ...
   <statements
xsi:type="statements:LocalVariableStatement">
    <variable name="Z">
     ...
    </variable>
   </statements>
   <statements
xsi:type="statements:LocalVariableStatement">
    <variable name="one">
     <typeReference
xsi:type="types:NamespaceClassifierReference">
      <classifierReferences>
       <target xsi:type="tom:DataType" href=

"//@classifiers.0/@members.0/@dataTypes.0"/>
      </classifierReferences>
     </typeReference>
     <initialValue xsi:type="tom:BackquoteTermBuilder"
```

```
      target=
"//@classifiers.0/@members.0/@dataTypes.0/@operators.0"/>
      <arguments xsi:type=
        "references:IdentifierReference" target=

"//@classifiers.0/@members.1/@statements.0/@variable"/>
    </variable>
   </statements>
      ...
  </members>
 </classifiers>
</containers:CompilationUnit>
```

Listing 11. XMI representation of class Main according to the extended metamodel

6.3. Adding, Removing and Modifying Refactoring

In this section, two modification scenarios are illustrated. The first scenario is related to limitations encountered when using a precondition-based approach to guarantee behavioral preservation. Soares et al. [27] evaluated refactoring engines like Eclipse JDT and NetBeans using SafeRefactore, a tool for checking behavioral changing. They reported many overly weak and overly strong preconditions. Overly weak preconditions are insufficient to ensure behavioral preservation whereas overly strong preconditions prevent refactorings, where some minor modifications to the code would enable them. Hence, developers must enforce the first type and relax the second. In section 2, we used an overly strong precondition for the remove class refactoring, which must be relaxed. The precondition prevents removing classes that are referenced internally or imported but never instantiated. A more flexible removing would delete this kind of referenced class, which can be achieved by changing the primitive analysis operation *isReferenced()* of Listing 4. Generally, relaxing an overly strong precondition requires the introduction of new code transformations which is not the case here. The new primitive operation *isReferenced()* is shown in Listing 12.

The second scenario requires the modification of some existing refactorings and the introduction of new QVT transformations. Consider the example of renaming the class Main of Listing 10 to Main2. As explained previously, the old implementation of this refactoring is no longer valid. Our prototype is extended to support renaming Tom files. The process comprises the following steps:

- Extend the infrastructure to manipulate Tom files (section 6.2);
- Add new QVT transformations to apply refactoring;
- Add the manipulating code to the RefactoringManager component. It loads and saves models (thanks to the TomParser and Printer) using the TomResourceFactory which automatically selects Tom files (*.t files). Additional infrastructure is needed to resolve cross-references between model elements since Tom programs can import

and include Tom constructs stored in separated files (e.g., *.gom files). Also, the code calls the *renameClass* OCL operation and invokes the new QVT transformations. The *renameClass* precondition is extended to prevent the refactoring if a package whose name is same as the new name exists in the parent container of the Tom file to be renamed.

The QVT transformations do the following:

– Find the corresponding package (in the parent package of the file to be renamed) and rename it, if exists;
– Find the corresponding Java file (in the parent package of the file to be renamed) and rename it, if exists;
– Get the Java import elements whose namespaces respect Tom conventional form (i.e., *packagename/filename/modulename*) and change the corresponding namespace (i.e., from *filename* to *newfilename*), if exist;
– Update all references to the class. This includes Tom-specific references such as the *implement{classreference}* reference of the *TypeTerm* construct;
– Rename the class contained in the file and then rename the file.

```
Context
  java::classifiers::Class::isReferenced():Boolean
body:
  java::types::NamespaceClassifierReference.
  allInstances()->
  exists(ncr|
          not (ncr.getContainingConcreteClassifier().
               oclIsTypeOf(java::classifiers::Class) and
               ncr.getContainingConcreteClassifier().
               oclAsType(java::classifiers::Class)= self
               ) and
          ncr.classifierReferences->
          exists(cr|

cr.target.oclIsTypeOf(java::classifiers::Class) and

cr.target.oclAsType(java::classifiers::Class)= self
                 )
        ) or
  java::references::IdentifierReference.allInstances()->
  exists(ir|
          not (ir.getContainingConcreteClassifier().
               oclIsTypeOf(java::classifiers::Class) and
               ir.getContainingConcreteClassifier().
               oclAsType(java::classifiers::Class)= self
               ) and
          ir.target.oclIsKindOf(java::members::Member)and
          ir.target.oclAsType(java::members::Member).
          getContainingConcreteClassifier().oclIsTypeOf
          (java::classifiers::Class) and
          ir.target.getContainingConcreteClassifier().
          oclAsType(java::classifiers::Class)= self
        ) or
  java::references::MethodCall.allInstances()->
  exists(mc|
          not (mc.getContainingConcreteClassifier().
               oclIsTypeOf(java::classifiers::Class) and
               mc.getContainingConcreteClassifier().
               oclAsType(java::classifiers::Class)= self
               ) and

mc.target.oclIsTypeOf(java::members::ClassMethod) and

mc.target.oclAsType(java::members::ClassMethod).
          getContainingConcreteClassifier().
          oclAsType(java::classifiers::Class)= self
        )
```

Listing 12. Relaxed primitive analysis operation isReferenced()

6.4. Discussion

Code generation is a powerful technique used in model-driven software development. Bringing this technique into the refactoring world is one of the objectives of this work. In the evaluation section, we applied several modification scenarios to our prototype to validate the extensibility of refactoring tools implemented based on our approach. We selected the Tom language as a case study. Why Tom? Because it’s a very complex Java extension that requires the implementation of many complex parsers. We are talking about an extreme extension case.

We have learned from this experience that creating metamodels for the languages is not easy for developers who do not have a deep knowledge about them, and this goes for creating concrete syntaxes. Fortunately, refactoring tool developers fulfill this requirement. Also, resolving model references is not an easy task and creating resolvers requires skillful programming abilities; especially for complex language extensions. However, not all Java extensions are as complex as the Tom case; embedded languages with unique opening and closing tags can be easily integrated with JaMoPP and their resolving rules are simpler.

7. Related Work

Since the early days of refactoring, it has been clear that automation is needed. Several approaches have been proposed for facilitating the implementation of refactoring tools. Steimann et al. proposed constraint-based refactoring [28], [30], [31]. The idea is to describe a refactoring problem by constraints generated from the programs to be refactored using so-called constraint rules. The refactoring constraint language and framework REFACOLA is developed to help implementing constraint-based refactoring tools for any target language [29]. Schäfer et al. suggest the use of an intermediate representation of Java programs called J_L [26]. They attempt to make complex refactorings simpler to express and implement by introducing a more efficient solution to the issues of naming and accessibility. In a preceding work, Schäfer et al. employed a decomposition and modular approach to specify complex refactorings using micro-refactorings [25]. They proposed language restrictions and extensions to make refactorings easier to formulate. Schäfer' approach addresses the preservation of program behavior as a dependence edge preservation problem [23], [24], and uses constraints to control accessibility adjustments [26].

IDEs (e.g., Eclipse, NetBeans, IntelliJ IDEA) played a big role in the popularization and utilization of refactorings. The Eclipse JDT (i.e., Java Development Tool), for example, includes a wide range of automatic refactorings [33]. As a JDT generalization, an API called LTK (The Language Toolkit) is introduced to support automated refactorings implementation in eclipse-based IDEs for other programming languages (e.g., C++, Fortran). Like most IDEs, eclipse uses an AST built by a compiler front end (Eclipse Compiler for Java or ECJ) for analyzing and changing the code. When it comes to customizing and extending its refactoring tool (to fix bugs, add refactorings, customize refactorings, add language features,…), choices are limited by the IDE's existing parser and AST (i.e., the JDT API).

In his work, Overbey proposes the use of a language-independent library to generate most of the code needed to implement the refactoring tool components (parser, AST, etc.) [20]. His approach focuses on using grammar to generate ASTs with a very rich API, and gives the tool developer the possibility to customize the structure of ASTs (and thus the parser) by annotating the grammar. However, developers are obliged to work with the author's toolkit (Ludwig which is a grammar-based code generator with an EBNF parser, and the Rephraser Engine which is a language-independent library).

Unlike the techniques mentioned above, our solution uses a standard-based approach to develop, extend, and maintain refactoring tools. Rather than manipulating the abstract syntax tree, we manipulate models using an Ecore-based representation of the abstract syntax. Similar to Overbey's approach, components underlying the refactoring tool are generated from a grammar, but using a language workbench in a model-driven fashion. Code analysis for precondition checking and code transformation are implemented separately, with OCL and QVT respectively (standards). This separation reduces the effort required in the tool maintenance and extension.

Similarly, Liang introduces a model-driven approach for precondition checking [10]. He uses OCL to specify refactoring preconditions for C++ programs. However, and besides the fact that he didn't cover the code transformation part, his implementation relies on a Component-Based Tool Development technique where the Eclipse CDT API (i.e., the eclipse parser and AST for C++) is used to extract artifacts from the source code, then models representing the program are constructed against the language metamodel by mapping the extracted AST elements to the corresponding model elements. He claims that programmers should be able to specify their own preconditions and he bases his approach on this hypothesis. The problem is that many programmers are neither familiar with metamodeling and declarative languages such as OCL nor experts in refactoring precondition specification. Our approach targets refactoring tool developers who are familiar with model-driven techniques.

8. Conclusion

In this paper, we have used model-driven techniques and a language workbench to develop and extend a Java refactoring tool. The presented approach has a number of advantages. The language metamodel is considered to be part of the tool's documentation, which supports the tool's code comprehension. This is because the OCL preconditions and the QVTO transformations are specified against the metamodel. Thus, understanding tool behavior and outcome by developers, maintainers, or even users becomes easier, especially for those who are familiar with modeling languages.

The above conclusion is fortified by the fact that our approach is based on standards. The separation of behavior preservation checking (i.e., preconditions checking) from code transformation enables better extension of the tool. One can accommodate new refactoring preconditions (e.g., to handle new language features) without affecting what the transformation actually does and vice versa. Thanks to EMFText, extending the language (i.e., Java) can be easily done by extending the language metamodel and the text syntax specification from which the tooling underlying the refactoring tool can be regenerated.

In future we plan to support our prototype to undo refactorings. Eclipse is used to provide a user interface. However, improvements (e.g., preview and error reporting) are required. The verification and the evaluation of the implemented model transformations to improve the reliability and the quality of our prototype are subject to future work. There are different verification techniques of model transformations that can be used [2], [9]. In the same context, we intend to run more test suites on refactored code from open-source projects to check behavioral changes. Several techniques and tools can be of help to detect bugs and validate the implemented refactorings [4], [11], [27].

Like other model-driven approaches, scalability is an important concern, especially in the context of refactoring. A refactoring tool is supposed to handle complex and large-scale software with acceptable performance. Modeling frameworks like EMF tried to solve the growing size of models issue by introducing mechanisms such as lazy loading and databases for model persistence. Here, we exploited the lazy loading mechanism when we implemented the refactoring manager component (section 5.2) which improved our prototype performance. However, further work is needed to improve these solutions. OCL and QVTO code optimization along with highly designed metamodel for the language are required to achieve desirable performance. In this work, only extensibility is evaluated. Doing the comparison between our prototype and other refactoring engines (e.g., Eclipse JDT) in term of performance is also subject to future work.

References

1. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.): RTA 2007. LNCS, Vol. 4533. Springer, Heidelberg, 36–47. (2007)
2. Calegari, D., Szasz, N.: Verification of Model Transformations: A Survey of the State-of-the-Art. Electronic Notes in Theoretical Computer Science, Vol. 292, 5–25. (March 2013)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. (1999)
4. Gligoric, M., Behrang, F., Li, Y., Overbey, J., Hafiz, M., Marinov, D.: Systematic Testing of Refactoring Engines on Real Software Projects. In: Castagna, G. (ed.): Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13). LNCS, Vol. 7920. Springer-Verlag, Berlin Heidelberg, 629-653. (July 2013)
5. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: EMFText and JaMoPP - Tool Presentation. In: Vogel, R. (ed.): 5th European Conference on Model Driven Architecture- Foundations and Aplications (ECMDA-FA'09): Proceedings of the Tools and Consultancy Track, 73-77. (June 2009)
6. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In Proceedings of the 5th European Conference on Model Driven Architecture- Foundations and Aplications (ECMDA-FA'09). LNCS, Vol. 5562. Springer, 114–129. (June 2009)
7. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), Ser. LNCS, Vol. 5969. Springer, 374-383. (March 2010)
8. Izquierdo, J., Cuadrado, J., Molina, J.: Gra2MoL: A Domain Specific Transformation Language for Bridging Grammarware to Modelware in Software Modernization. In MODSE: Workshop on Model-Driven Software Evolution, Spain. (2008)
9. Kolahdouz-Rahimi, S., Lano, K., Pillay, S., Troya, J., VanGorp, P.: Evaluation of Model Transformation Approaches for Model Refactoring. Science of Computer Programming, Vol. 85, Part A, 5-40. (1 June 2014)
10. Liang, Y.: Code Refactoring Under Constraints. PhD thesis, University of Alabama, Tuscaloosa, AL,USA. (2011)
11. Mongiovi, M., Gheyi, R., Soares, G., Teixeira, L., Borba, P.: Making Refactoring Safer Through Impact Analysis. Science of Computer Programming, Vol. 93, 39-64. (November 2014)

12. Moreau, P.-E., Ringeissen, C., Vittek, M.: A Pattern Matching Compiler for Multiple Target Languages. In: Hedin, G. (ed.): CC 2003. LNCS, Vol. 2622. Springer-Verlag, Berlin Heidelberg, 61–76. (2003)
13. Mur, G., Kersten, M., Findlater, L.: How are Java Software Developers Using the Eclipse IDE?. IEEE Software, Vol. 23, No. 4, 76-83. (July 2006)
14. Murphy-Hill, E., Parnin, C., Black, A.P.: How We Refactor, and How We Know It. IEEE Transactions on Software Engineering, Vol. 38, No. 1, 5-18. (January-February 2012)
15. Object Management Group (OMG), Meta Object Facility (MOF) 2.0 Query/ View/Transformation Specification, Version 1.1, http://www.omg.org/spec/QVT/1.1/PDF. (January 2011)
16. Object Management Group (OMG), Object Constraint Language (OCL), Version 2.3.1, http://www.omg.org/ spec/OCL/2.3.1/PDF. (January 2012)
17. Object Management Group (OMG), OMG Model Driven Architecture, http://www.omg.org/ mda/.
18. Object Management Group (OMG), Meta Object Facility (MOF) Core Specification, Version 2.4.2, http://www.omg.org/spec/MOF/2.4.2/PDF. (April 2014)
19. Opdyke, W. F.: Refactoring Object-oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA. (1992)
20. Overbey, J.: Thesis Overview: A Toolkit for Constructing Refactoring Engines. Journal of Computer Science and Technology, Vol. 12, No. 3, 140-142. (2012)
21. Pawlak, R.: Spoon: Compile-time Annotation Processing for Middleware. IEEE Distributed Systems Online, Vol. 7, No. 11, 1. (November 2006)
22. Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, Illinois, USA. (1999)
23. Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In: Kiczales, G. (ed.): Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA’08), ACM Press, New York. (2008)
24. Schäfer, M., Verbaere, M., Ekman, T., de Moor, O.: Stepping Stones Over the Refactoring Rubicon–Lightweight Language Extensions to Easily Realise Refactorings. In: Drossopoulou, S. (ed.): ECOOP 2009- Object-Oriented Programming. LNCS, Vol. 5653. Springer, Berlin Heidelberg, 369-393. (2009)
25. Schäfer, M., de Moor, O.: Specifying and Implementing Refactorings. In Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10). ACM, New York, NY, USA, 286-301. (2010)
26. Schäfer, M., Thies, A., Steimann, F., Tip, F.: A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. IEEE Transactions on Software Engineering, Vol. 38, No. 6, 1233-1257. (November-December 2012)
27. Soares, G., Gheyi, R., Massoni, T.: Automated Behavioral Testing of Refactoring Engines. IEEE Transactions on Software Engineering, Vol. 39, No. 2, 147-162. (February 2013)
28. Steimann, F., Thies, A.: From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In: Drossopoulou, S. (ed.): ECOOP 2009- Object-Oriented Programming. LNCS, Vol. 5653. Springer, Berlin Heidelberg, 419-443. (2009)
29. Steimann, F., Kollee, C., von Pilgrim, J.: A Refactoring Constraint Language and Its Applic-ation to Eiffel. In: Mezini, M. (ed.): ECOOP'11. LNCS. Vol. 6813, Springer, Berlin Heidelberg, 255-280. (2011)
30. Steimann, F., von Pilgrim, J.: Constraint-Based Refactoring with Foresight. In Proceedings of ECOOP'12. LNCS, Vol. 7313. Springer, Heidelberg, 535-559. (2012)
31. Steimann, F., von Pilgrim, J.: Refactorings Without Names. In Proceedings of the 27th IEEE/ ACM International Conference on Automated Software Engineering (ASE), 290-293. (2012)
32. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. 2nd Edition, Addison Wesley Professional. (2008)
33. The Eclipse foundation, http:// www. eclipse.org/org/.

34. The Eclipse foundation, The Eclipse OCL website, http://www.eclipse.org/modeling/mdt/ocl/.
35. The Eclipse foundation, The Eclipse QVTO website, https://www.eclipse.org /mmt/QVTO/
36. Xing, Z., Stroulia, E.: Refactoring practice: How it is and How it Should be Supported—an Eclipse Case Study. In Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06), Washington, DC, USA, 458-468. (2006)

Sohaib Hamioud received his Master degree in Computer Science from the University of Annaba, Algeria in 2010. He is a PhD student and a member of the SAFA team in LISCO laboratory at the University of Annaba. His research interests include model-driven engineering, domain specific languages, software architecture and software engineering.

Fadila Atil is a Professor in the Department of computer science at the University of Annaba. She received her PhD degree in 2007 from the same university. She participated as Manager or Member in several research projects. Currently, she is Manager of the project SAFAXY-1 and Team Manager of SAFA team in LISCO laboratory at the University of Annaba. Her research interests include software architecture, evolution and reuse in software engineering.

Received: October 25, 2014; Accepted: April 04, 2015