IDEA GROUP PUBLISHING
701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com
ITB12817

This paper appears in the book, *Emerging Trends and Challenges in Information Technology Management, Volume 1 and Volume 2* edited by Mehdi Khosrow-Pour © 2006, Idea Group Inc.

Specifying Refactorings as Metamodel-Based Transformations

Claudia Pereira, INTIA, Departamento de Computación Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil - Argentina, cpereira@exa.unicen.edu.ar

Liliana Favre, INTIA - Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil - Argentina, & CIC, Buenos Aires, lfavre@exa.unicen.edu.ar

ABSTRACT

The Model Driven Architecture (MDA) is facing a paradigm shift from object-oriented software development to model-centric development. MDA distinguishes at least three different kinds of models: Platform Independent Model (PIM), Platform Specific Model (PSM) and Implementation Specific Model (ISM). With the MDA approach, some crucial points are the refactoring techniques that allow model transformations leaving their behavior unchanged but enhancing some non-functionality quality factors. In this paper we propose a uniform treatment of refactorings at levels of PIMs, PSMs, and ISMs. We define refactorings as metamodel-based transformation contracts that can be used to validate and test transformations.

1 INTRODUCTION

The Model Driven Architecture (MDA) is an initiative proposed by the Object Management Group (OMG) to model-centric software development (MDA, 2003). MDA promotes the creation of abstract models that are developed independently of particular platforms and then automatically transformed by tools into models or code for specific platforms or technologies. It distinguishes at least three different kinds of models: Platform Independent Model (PIM), Platform Specific Model (PSM) and Implementation Specific Model (ISM). A PIM is a model that contains no reference to the platforms that are used to realize it. A PSM describes a system in the terms of the final implementation platform e.g., .NET or J2EE. An ISM refers to components and applications.

A Model Driven Development (MDD) is carried out as a sequence of model transformations. We can distinguish two types of transformations to support model evolution from PIMs to ISMs: refinements and refactorings. A refinement is the process of building a more detailed specification that conforms to another that is more abstract. On the other hand, a refactoring means changing a model leaving its behavior unchanged, but enhancing some non-functionality quality factors such as simplicity, flexibility, understandability and performance.

Refactoring is a crucial point in model evolution. Although the most effective forms of refactorings are at the design levels (e.g, PIMs or PSMs), MDA-based Case tools provide limited facilities for refactoring only on source code through an explicit selection made by the designer (CASE UML, 2005). In this light, we propose a metamodeling technique to define refactorings at different abstraction levels in a uniform way. A transformational system based on behavior-preserving model-to-model transformations was defined. To reason about correctness and robustness we propose to specify refactorings as OCL contracts that are based on metamodels capturing common properties to a family of refactorings.

This paper is structured as follows. Section 2 provides some background on refactoring in the MDD context. Section 3 exemplifies rules to restructure models at levels of PIMs, PSMs and ISMs. Section 4 discusses how to specify model-to-model transformations as OCL contracts. Section 5 considers related work. Finally, in Section 6 conclusions and future work are given.

2 REFACTORING AND MDD

Key to MDA is the importance of models in the software development process. MDA defines a framework that separates the specification of the system functionality from its implementation on a specific platform. MDA distinguishes different kinds of models:

- Platform Independent Model (PIM), a model with a high level of abstraction that is independent of any implementation technology.
- Platform Specific Model (PSM), a tailored model to specify the system in terms of the implementation constructs available in one specific implementation technology.
- Implementation Specific Model (ISM), a description (specification) of the system in source code.

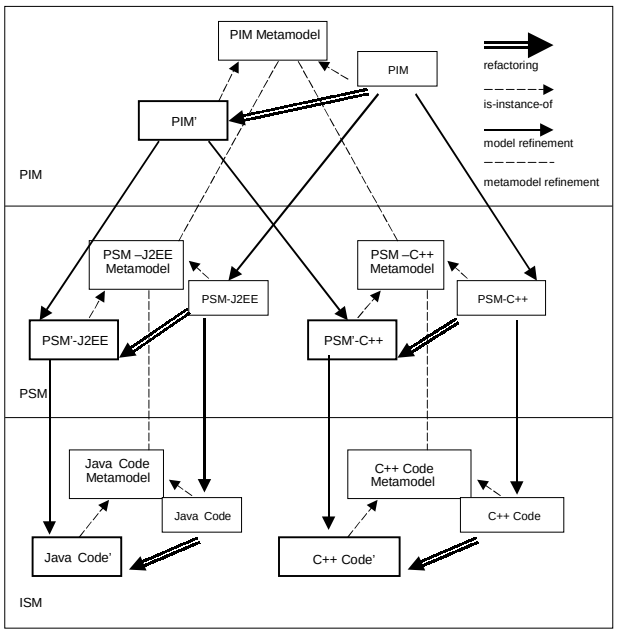
A model driven development is carried out as a sequence of model transformations that includes at least the following steps: construct a PIM that provides a computing architecture independent of specific platforms; transform the PIM into one or more PSMs, and derive code directly from the PSMs (MDA, 2003; Kleppe et al., 2003).

One of the main key issues behind the Model-Driven Development is that all artifacts generated during software development are represented using metamodeling languages. In MDA, they are expressed as a combination of UML class diagrams and OCL constraints (UML, 2005; OCL, 2005). The 4 main core metamodeling constructs are classes, binary associations, data types and package.

The transformations between models are described relating each element of the source model to one or more elements of the target model at metamodel level. In other words, relating the metaclass of the element of the source model with the metaclasses of the element of the source model. The models to be transformed and the resulting models of the transformations will be instances of the corresponding metamodel. Fig. 1 shows the relations between PIMs, PSMs and ISMs. The following types of transformations can be distinguished:

- Refactoring. It is applied to a model in a given level generating a new restructured model in the same level (PIM to PIM, PSM to PSM, ISM to ISM).
- PIM to PSM Refinement. It describes how a PIM that is an instance of a UML-Metamodel is transformed into a PSM that is an instance of a specialized metamodel for a specific platform.
- PSM to ISM Refinement. It describes how a PSM is transformed into code (which is an instance of UML Metamodel for a platform and specific language technologies).

Figure 1. Refactoring in MDD



3 MDA-BASED REFACTORINGS

Refactoring is a powerful technique when it is repeatedly applied to a model to obtain another one with the same behavior. A transformational system for refactoring UML static models is proposed. The goal is to provide support for small refactorings by applying semantics-preserving transformation rules. Transitions among versions are made according to precise rules based on the redistribution of classes, variables, operations and associations across the diagram in order to facilitate future adaptations and extensions.

We define a library of refactorings that classifies them at PIM, PSM and ISM levels. Section 3.1 informally describes the system of transformation rules for refactoring models at different abstraction levels.

3. 1 Transformation Rules

This section shows examples of transformation rules applied at different levels, PIMs (3.1.1), PSMs (3.1.2.) and ISMs (3.1.3.). We use textual and diagrammatic descriptions to describe each example of refactoring.

3. 1. 1 Examples of PIM Refactorings

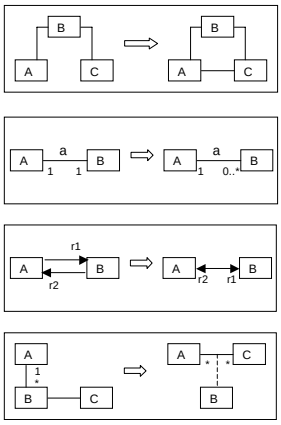
Adding a transitive association: Given an association between classes *A* and *B* and an association between classes *B* and *C*, an association may be derived between *A* and *C*, determining the appropriate association type, the multiplicities and the navigability of each association end. (Whittle, 2002)

Substitution of an association: Given an association *a*, it may be substituted with a less constrained association of the same name, i.e., in any association *a*, an association-end *E* with multiplicity *mult1* may be substituted with an association-end *E* with multiplicity *mult2*, where *mult1* $\subseteq$ *mult2*. (Evans, 1998)

Joint of unidirectional associations: Two unidirectional associations with navigability in opposite direction may be joined in a plain bidirectional one (Kollmann & Gogolla, 2001).

Adding an association class: Given a class that associates with other two classes, with association ends with the other classes with multiplicity equal to 1, it may be transformed to an association class.

Example 3. 1. 1



3. 1. 2 Examples of PSM Refactorings

Folding: It joins two classes which have a direct inheritance relationship obtaining a new class gathering the behavior of both. The goal is to reduce the level of a class hierarchy in those cases where there is no particular interest in the behavior of a base class.

Abstraction: It divides the behavior of a class generating two classes which maintain a direct inheritance relationship. By the application of this rule, a new base class can abstract the more general behavior identified inside another class.

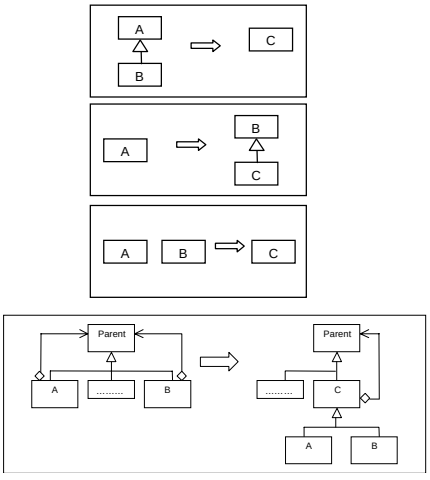
Union: It gathers two classes without inheritance relationship to each other in a new one. This rule can be useful to group behavior and to reduce the multiple inheritances.

Extract Composite: It extract a superclass that implements the *Composite*, when subclasses in a hierarchy implement the same *Composite*. (Kerievsky, 2004)

3. 1. 3 Examples of Code Refactorings

Replace Temp with Query: Given a temporary variable in a method body that hold in the result of an expression, it may be replaced with a query method. The expression is extracted into a method. All references to the expression are replaced. The new method can then be used in other methods. (Fowler, 1999)

Example 3. 1. 2



Example 3. 1. 3

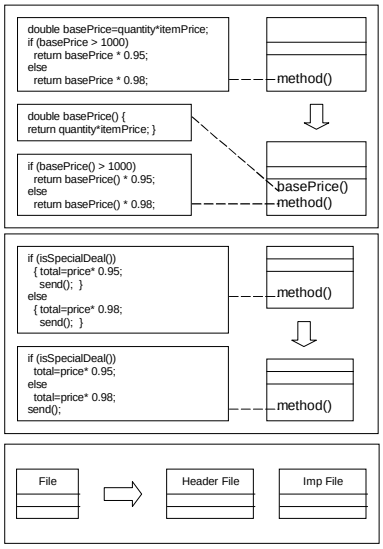
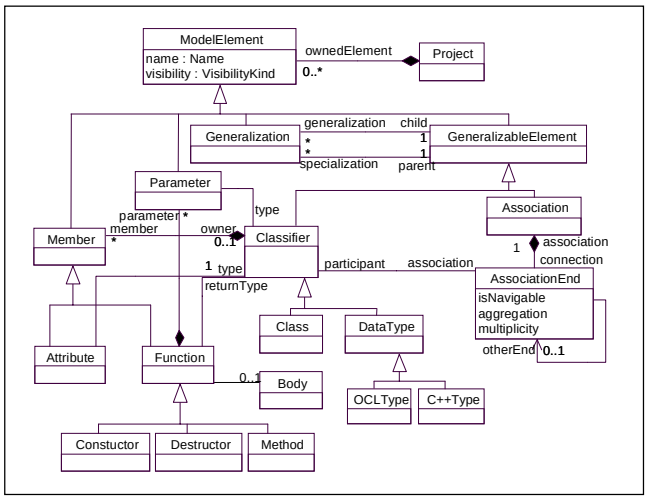


Figure 2. A simplified C++ metamodel



Consolidate Duplicate Conditional Fragments: When a same fragment of code is in all branches of a conditional expression, the fragment of code may be moved outside the expression. This makes clearer what varies and what stays the same. (Fowler, 1999)

Modularization: Given a file containing interface descriptions and implementations, the rule generates a header file with the interface descriptions and an implementation file.

4 METAMODELING APPROACH FOR REFACTORINGS

A metamodeling approach to define transformation rules at different abstraction level is proposed. Metamodel transformations impose relations between a source metamodel and a target metamodel both represented as UML class diagrams annotated in OCL. The source metamodel defines the family of source models to which refactorings can be applied and the target metamodel characterizes the models that are generated by conforming an OCL contract.

Refactorings are described by transformation rules that consist of a name, a set of parameters, a precondition and a postcondition. Each parameter is a metamodel element. The precondition, which deals with the state of the model before the transformation, states relations at the metamodel level between the elements of the source model. The postcondition, which deals with the state of the model after the transformation, states relations at metamodel level between the elements of the source model and a target model. Rules can also include local operations that are used in preconditions and postconditions. The application of these rules can generate new elements on the model, modify or remove existing ones.

The restructuring rules are basic units of transformation, i.e., starting from them, particular sequences can be built to solve situations presented in a model which is wanted to improve. These predefined sequences are denominated restructuring strategies and they were exemplified in (Pereira et al., 2004).

Next, we define the *Extract Composite* refactoring (see 3.1.2.) by using a C++ platform. In Fig. 2 we show a simplified C++ metamodel. The *Extract Composite* rule allows extracting a superclass that implements the *Composite*. Below, we partially show the *Extract Composite* transformation as an OCL contract. The *Extract Composite* refactoring consists of the following steps: create a *Composite*; make each child container (a class in the hierarchy that contains duplicate methods) a subclass of the *Composite* and move duplicated methods across the child containers to the *Composite* (Kerievsky, 2004). Comments explaining postconditions are attached following (see Box A).

5 RELATED WORK

The first relevant publication on refactoring was carried out by Opdyke (1992), showing how functionalities and attributes can migrate among classes, how classes can be joined and separated using a class diagram notation (subset of current UML). Roberts (1999) completed this work describing techniques based on refactoring contracts.

Fowler (1999) informally analyzes refactoring techniques on Java source code, explaining the structural changes through examples with class diagrams. Fanta & Rajlich (1998) and Fanta & Rajlich (1999) study refactoring of C++ code.

Several approaches provide support to restructure UML models. In (Gogolla & Ritchers, 1998) advanced UML class diagram features are transformed into more basic constructions with OCL constraints. Evans (1998) proposes a rigorous analysis technique for UML class diagrams based on deductive transformations. In (Sunyé et al., 2001) a set of refactorings is presented and how they may be designed to preserve the behavior of UML models is explained. Philipps & Rumpe (2001) reconsider existing refinement approaches to formally deal with the notions of behavior, behavior equivalence and behavior preservation. Whittle (2002) investigates the role of transformations in UML class diagrams with OCL constraints. Mens et al. (2003) provide an overview of existing research in the field of refactoring. Porres (2003) defines and implements model refactorings as rule-based transformations. Van Gorp et al. (2003) propose a set of minimal extensions to UML metamodel, which allows reasoning about refactoring for all common object-oriented languages. Thomas (2005) analyses the state of the art in refactoring and issues such as languages and tool impact on refactoring, refactoring as meta-programming, refactoring and persistent instances.

Tools are available to automate several refactoring aspects. For example, Guru (Moore, 1995) is a fully automated tool to restructure inheritance hierarchies of SELF objects preserving behavior. Smalltalk Refactoring Browser (Roberts et al., 1997) is an advanced browser for VisualWork which automatically carries out transformations which preserve behavior. There is a tendency to integrate refactoring tools into industrial software development environments. For example, Together ControlCenter (TogetherSoft, 2005) applies code refactoring on user requirements and IntelliJ IDEA (IntelliJ IDEA, 2005) comes fully equipped with refactoring tools.

Box A.

```
Transformation Extract Composite {
parameters
      source: C++ Metamodel:: Project
      target: C++ Metamodel:: Project
      parentClass: C++ Metamodel:: Class
      subclasses: Set (C++ Metamodel::Class)
local operations
      isConsistentWith (f1: Function, f2: Function): Boolean
      -- return true if f1 is consistent with f2:
      isConsistentWith (f1, f2) =
            -- f1 and f2 reference equivalent attributes,
            f1.referencedAttributes →size() = f2.referencedAttributes →size() and
            f1.referencedAttributes→forAll (a1 / f2.referencedAttributes→
                                                exists(a2     /equivalentAttributes(a1,a2)))
            -- f1 and f2 have the same number of parameter,
             f1.parameter → size() = f2.parameter → size() and
            -- the type of each formal parameter of f1 conforms to the type of the corresponding elements of f2,
            Sequence {1..(f1.parameter→size())} →
            forAll (index: Integer/  conformTo(f1.parameter→at(index).type,f2.parameter→at(index).type)) and
            -- the return type  of f1 conforms to the return type of f2 and
            conformTo(f1.returnType, f2.returnType)
            ……..
      isConsistentInAllSubclasses (f: Function): Boolean
      isConsistentInAllSubclasses (f) =
            subclasses → forAll(s/ s.member.oclIsTypeOf(Function) → exists(fun/
                  isConsistentWith(f , fun) or  isConsistentWith(fun, f) ))
      conformTo (c1: Classifier, c2: Classifier): Boolean
      -- return true if the classifier c1(that defines a type) conforms c2.
      conformTo (c1, c2) = (c1=c2) or (c1.allParents() → includes(c2))
      ……..
preconditions
      -- subclasses and parentClass are element of source model.
      source.ownedElement.oclIsTypeOf(Class) → includes (parentClass)  and
      source.ownedElement.oclIsTypeOf(Class) → includesAll (subclasses)  and

      -- subclasses collection contains subclasses of parentClass.
      subclasses → forAll( c/ c.generalization.parent → includes (parentClass))  and

      -- each class of subclasses has a binary association with parentClass whose association
      -- end tie to the subclass is an aggregation
      subclasses → forAll (c / c.association.association → exists (a / a.connection → size() = 2 and
            a.connection→exists (e/e.participant=c and e.aggregation=#aggregation)) )  and

      -- all subclasses have functions that can be factorized.
      subclasses → forAll (c/ c.member.oclIsKindOf(Function) → exists (f/
            isConsistentInAllSubclasses(f) ))  and

      -- subclasses collection  has at least two classes.
      subclasses → size() >= 2  and
 postconditions
       --in the target model exists a class, C, so that
       target.ownedElement.oclIsTypeOf(Class) → exists ( c /

          -- class C is created during Extract Composite transformation.
          c.oclIsNew()  and

          -- C has a parent class with:
          c.generalization.parent.oclIsTypeOf(Class) → exists(class/
              -- the same name of parentClass,
              class.name=parentClass.name and
              -- the same generalization class collection of parentClass,
      class.generalization.parent.oclIsTypeOf(Class) =
      parentClass.generalization.parent.oclIsTypeOf(Class) and
              -- the same members of parentClass,
              class.member=parentClass.member  and
              -- and the specialization class collection contains class C and those subclasses of
              -- parentClass that not belong to subclasses collection.
              class.specialization.child.oclIsTypeOf(Class) → includes(c) and
              class.specialization.child.oclIsTypeOf(Class) →
              includesAll(parentClass.specialization.child.oclIsTypeOf(Class) ->subclasses))  and

   -- C has factorized functions from subclasses.
   c.member.oclIsKindOf(Function) → forAll(f/ subclasses → forAll ( sub /
              sub.member.oclIsKindOf(Function) → exists ( fsub/ isConsistentWith(fsub, f))))  and

   -- C has factorized association end from subclasses.
   c.association → forAll( a/ subclasses → forAll ( sub /
              sub.association → exists ( asub/ equivalentAssociationEnd(asub, a))))  and

   -- C has factorized attributes from subclasses.
   ……..
   -- for each subclass of C, subc, there is a subclass of subclasses, sub, so that
   c.specialization.child.oclIsTypeOf(Class)  → forAll (subc/
        subclasses→ exists (sub/
            -- subc and sub have the same name,
            subc.name=sub.name  and

            -- the same child classes,
            subc.specialization.child.oclIsTypeOf(Class) = sub.specialization.child.oclIsTypeOf(Class)   and

            -- subc excludes functions equivalent with those that were factorized to class C,
            sub.member.oclIsKindOf(Function) → forAll( f/
                c.member.oclIsKindOf(Function)  → forAll ( fc/
                    if (isConsistentWith (f, fc) then
                    subc.member.oclIsKindOf(Function) → excludes (f) and
                            --invocations of f must be consistent with factorized function interface
                            ……..
                    else subc.member.oclIsKindOf(Function) → includes (f)
                    endif)
            -- subc excludes associations equivalent to factorized associations,
            sub.association → forAll(as/
                c.association → forAll (fas/ if (equivalentAssociation(a,fas)
                        then subc.association → excludes (as) and
                            --invocations of as must be consistent with factorized association end.
                            ……..
                        else subc.association → includes (as)
                    endif)

            -- subc excludes attributes equivalent with those that were factorized to class C
      ……..

      ) }
```

Kerievsky (2004) presents a method for pattern-directed refactorings. Long, Jifeng and Liu (2005) formalize Fowler´s refactorings as refinement laws in a relational calculus.

6 CONCLUSIONS AND FUTURE WORK

This work presents a rigorous approach for the refactoring on UML static models at levels of PIMs, PSMs and ISMs. Our focus is on behavior-preserving model-to-model transformations. The main contribution of this paper is proposing a classification of refactorings at PIM, PSM and ISM levels and a metamodeling-based refactoring process that allows users to define refactorings step-by-step across PIM, PSMs and code levels in a uniform way. Although the set of rules allows doing quite a number of interesting refactorings, it is limited since it does not focus on transformations that involve different UML views.

In a Model Driven Development different tools could be used to validate/verify models at different abstraction levels (PIMs, PSMs, or implementations). In this direction we propose to formalize UML/OCL metamodels and refactorings by using the metamodeling notation NEREUS that is independent of any formal language and can be translated to specific ones. A detailed description may be found at Favre (2005).

To demonstrate the feasibility of this approach, a prototype assisting in the refactoring on object-oriented hierarchies in C++ was implemented. The prototype implements a small, rather powerful, set of basic transformation rules (folding, abstraction, union, factoring). In this approach, mechanical tasks perform model transformations, verify conditions of transformation rules and keep track of the development process (Pereira et al, 2004). The prototype could be refined to be a practical tool for MDA-based refactoring.

REFERENCES

CASE UML (2005). Available: www.objectsbydesign.com/tools/umltools_byCompany.html

Evans, A. (1998). Reasoning with UML Class Diagrams. Proceedings of 2nd Workshop on Industrial Strength Formal Specification Techniques. Available: www.cs.york.ac.uk/puml/papers/evanswift.pdf

Fanta, R.; Rajlich, V. (1998). Reengineering an Object Oriented Code. Proceedings of IEEE International Conference on Software Maintenance (ICSM'98), 238-246.

Fanta, R.; Rajlich, V. (1999). Reestructuring Legacy C Code into C++. Proceedings of IEEE International Conference on Software Maintenance (ICSM´99), 77-85.

Favre, L. (2005). A Rigorous Framework for Model Driven Development. In: T. Halpin, J. Krogstie and K. Siau (Eds.). Proceedings of CAISE´05 Workshops. EMMSAD´05 Tenth International Workshop on Exploring Modeling Method in System Analysis and Design Porto, Portugal: FEUP Editions, 505-516.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Programs. Addison-Wesley.

Gogolla, M.; Richters, M. (1998). Transformation Rules for UML Class Diagrams. Proceedings of UML' 98 Workshop, Springer, Berlin, 92-106.

IntelliJ IDEA (2005). Available: www.jetbrains.com/idea/

Kerievsky, J. (2004). Refactoring to Patterns. Addison-Wesley.

Kleppe, A.; Warmer, J.; Bast, W. (2003). MDA Explained. The Model Driven Architecture: Practice and Promise, Addison-Wesley.

Kollmann, R.; Gogolla, M. (2001). Application of UML Associations and Their Adornments in Design Recovery. Proceedings of 8th Working Conference on Reverse Engineering (WCRE 2001), IEEE, Los Alamitos. Available: doi.ieeecomputersociety.org/10.1109/WCRE.2001.957812

Long, Q.; Jifeng, H.; Liu, Z. (2005). Refactoring and Pattern-directed Refactoring: A Formal Perspective. Technical Report 318, UNU-IIST, P.O.Box 3058, Macau.

MDA (2003). MDA Guide Version 1.0.1 Available: www.omg.org/mda

Mens, T.; Demeyer, S.; Du Bois, B.; Stenten, H.; Van Gorp, P. (2003). Refactoring: Current Reasearch and Future Trends. Proceedings

of Third Workshop on Language Descriptions, Tools and Applications (LDTA 2003), 120–130.

Moore, I. (1995). Guru–A tool for Automatic Restructuring of Self Inheritance Hierarchies, TOOLS USA 1995. Available: http://selfguru.sourceforge.net/guru.pdf

OCL (2005). UML 2.0 OCL Specification. OMG Adopted Specification ptc/03-10-14. Available:www.omg.org

Opdyke, W. (1992). Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois, Urbana-Champaign.

Pereira, C.; Favre, L.; Martinez, L. (2004). Refactoring UML Class Diagrams. Proceedings of 2004 Information Resources Management Association International Conference (IRMA 2004), New Orleans, Louisiana, USA, 506-509.

Philipps, J.; Rumpe, B. (2001). Roots of refactoring, Proceedings of 10th OOPSLA Workshop on Behavioral Semantics, Florida, USA. Available: www4.in.tum.de/~philipps/pub/oopsla01.pdf

Porres, I. (2003). Model Refactorings as Rule-Based Update Transformations. Proceedings of <<UML 2003>>, Lecture Notes in Computer Science 2863, Springer Verlag.159-174.

Roberts, D.; Brant, J.; Johnson, R. (1997). A refactoring tool for Smalltalk, Theory and Practice of Object Systems. Available: http://st-www.cs.uiuc.edu/~droberts/tapos.pdf

Roberts, D. (1999). Practical Analysis for Refactoring, PhD thesis, University of Illinois.

Sunyé, G.; Pollet, D.; LeTraon; Jézéquel, J. (2001). Refactoring UML Models, in Proc. UML 2001, Lecture Notes in Computer Science 2185, Springer-Verlag, 134-138.

Thomas, D. (2005). Refactoring as Meta Programming? Journal of Object Technology. Vol.4, no.1, January-February 2005, 7-11. Available: http://www.jot.fm/issues/issue_2005_01/column1

TogetherSoft, ControlCenter (2005). Available: www.togethersoft.com

UML (2005). UML 2.0 Superstructure Specification. OMG Adopted Specification: ptc/03-08-02 Available: www.omg.org.

Van Gorp, P.; Stenten, H.; Mens, T.; Demeyer, S. (2003). Towards automating source-consistent UML Refactorings. Proceedings of <<UML 2003>>, Lecture Notes in Computer Science 2863, Springer Verlag, 144-158.

Whittle, J. (2002). Transformations and Software Modeling Languages: Automating Transformations.in UML. Proceedings of <<UML 2002>>-The Unified Modeling Language. Lecture Notes in Computer Science 2460 (eds. J. Jezequel; H. Hussman) Springer-Verlag, 227-241.