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

ITP5002

Refactoring UML Class Diagram

Claudia Pereira
INTIA, Facultad de Ciencias 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 (Comisión de Investigaciones Científicas de la Provincia de Buenos Aires), lfavre@exa.unicen.edu.ar

Liliana Martinez
INTIA, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina,
lmartine@exa.unicen.edu.ar

ABSTRACT

Refactoring is now seen as an essential activity during software development and maintenance. In this paper, we describe a "rules + strategies" approach that allows the refactoring on UML static models together with OCL contracts. Our focus is on behavior-preserving model-to-model transformations. We also describe an experimental tool prototype to restructure object-oriented hierarchies.

1- INTRODUCTION

Refactoring is a change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality factors such as simplicity, flexibility, understanding and performance. Refactoring is now seen as an essential activity for handling software evolution. When extreme programming is used, it is often necessary to restructure already existing models and code (Beck, 2000). Also, legacy systems require refactoring in order to make it more understandable to future changes in flexible requirements.

UML CASE tools provide limited facilities for refactoring on source code through an explicit selection made by the designer. However, it will be well worth thinking about refactoring at the design level. The advantage of refactoring at UML level is that the transformations do not have to be tied to the syntax of a programming language. This is relevant since UML is designed to serve as a basis for code generation with the new Model Driven Architecture. The current UML 1.5 metamodel is insufficient to maintain the consistency between restructured design models, various design views and implementations. This situation might change in the future since UML evolves to version 2.0 (OMG, 2003).

In this paper, we are concerned about refactoring on UML class diagrams and traceability of changes in a model. We propose a transformational system for refactoring UML static models based on the "rules + strategies" approach. The goal of this transformational system is to provide support for small refactorings by applying semantics-preserving transformation rules. The basic idea is to obtain a model with the same behavior. Transitions between 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. During the transformation process, we need strategies to guide the application of transformation rules and which allow us to construct UML diagrams with improved quality factors. To demonstrate the feasibility of this approach, a tool prototype that assists in the refactoring of object oriented hierarchies was implemented.

This paper is structured as follows. In Section 2, related works are given. Section 3 presents the transformation system: some definitions and a set of rules and strategies to restructure classes and associations. Section 4 presents an example and Section 5 describes an experimental refactoring prototype. Finally, Section 6 concludes and discusses future work.

2- 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 join and separate using a class diagram notation (subset of current UML). Roberts (1999), describing techniques based on refactoring contracts, completed this work.

Fowler et al. (1999) informally analyze 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 model. 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 model is explained. Philipps & Rumpe (2001) reconsider existing refinement approaches as a way 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. Demeyer et al. (2002) 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.

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, 2003) applies code refactoring on user requirements.

3- THE TRANSFORMATION SYSTEM

3.1. Definitions

We describe those basic notions and notations of UML static models that we need for the remainder of the paper.

A UML class diagram is represented in terms of classes and associations:

This conference paper appears in the book, Innovations Through Information Technology, edited by Mehdi Khosrow-Pour. Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

classDiagram = (classes, associations) where

classes = set of *classes* belonging to *classDiagram*

associations = set of *associations* belonging to *classDiagram*

class = (className, P, A, OP, I, AS,H) where

P= set of *formal parameters*

A= {(N, T, A) / N= attribute name,
T = attribute type,
A = access type (private/protected/public) }

OP = {(N, A, P, R, Pre, Pos) /
N = operation name,
A = access type,
P = {(Np, Tp)/ Np = parameter name,
Tp = parameter type },
R = return type,
Pre = {pre/ pre = OCL constraint},
Pos = {pos / pos = OCL constraint } }

I= {I / I = OCL constraint }
AS= {(assocName, className)/ assocName = association name,
className = name of the associated class }

H= {(P, A, Pa) / P = super class name,
A= inheritance type (private/protected/public),
Pa= set of parameters}

association = (assocName, $assocEnd_1$, $assocEnd_2$, OclConstraints)
assocEnd = (className, rolName, multiplicity, assocType, visibil ity, navigability)

We assume the following functions:

set-attr: OP -> A
maps each $m \in$OP onto a set of attributes directly or indirectly referenced in *m*
set-oper: OP -> OP
maps each $m \in$ OP onto a set of operations directly or indirectly referenced in *m*
set-assoc: OP -> AS
maps each $m \in$ OP onto a set of associations directly or indirectly referenced in *m*
Given two operations *m1* and *m2* belonging to different classes
equiv-oper **: O x O -> {true, false}**
determine if *m1* and *m2* are equivalent or not.
Given two attributes *a1* and *a2* belonging to different classes
equiv-attr: **A x A -> {true, false}**
determine if *a1* and *a2* are equivalent or not.

3.2. Transformation Rules

This section presents transformation rules on UML/OCL class diagrams. First, a kernel of rules for refactoring classes is described. Next, a number of examples of transformation rules for associations is presented.

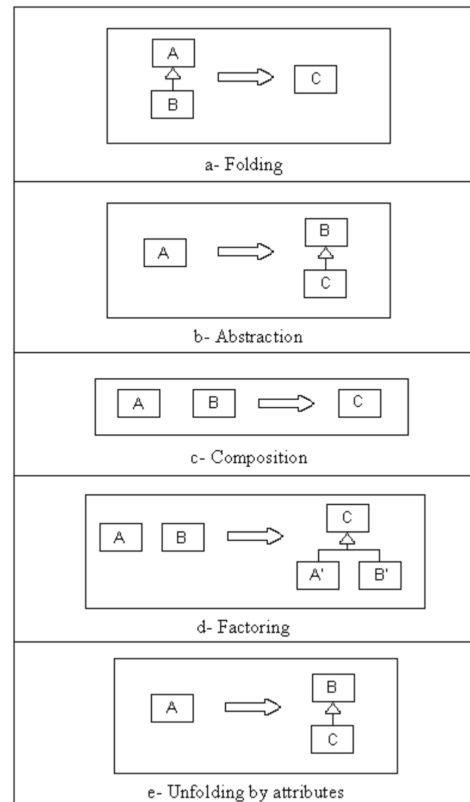
3.2.1. Refactorings: Dealing with inheritance

We define a set of transformation rules on UML static diagrams that allows transforming a hierarchy of classes. The transformations preserve the consistency and functional equivalence of the resulting hierarchy. Figure 1 shows the main rules.

Folding:

It joins two classes which have a direct inheritance relationship obtaining a new class gathering the behavior of both. The goal of this rule 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, either because it is an abstract class or because the amount of operations of the class does not justify having another level in the hierarchy.

Figure 1. Refactoring Class Hierarchies



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.

Composition:

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.

Factoring:

It factors equivalent operations in a new base class starting from two classes without inheritance relationship to each other. This rule eliminates duplicated operations and attributes.

Unfolding by attributes:

It divides the behavior of a class, generating two classes which maintain a direct inheritance relationship. Such classes arise from carrying out a partition of the attributes in two disjoint subsets. This rule is useful when operations do not reference simultaneously to all the attributes, but only make reference to some of them.

3.2.2. Refactoring Strategies

The restructuring rules are basic units of transformation, i.e., starting from them particular sequences can be built to solve situations presented in a hierarchy which is wanted to improve. These predefined sequences are denominated restructuring strategies. They allow applying

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

a sequence of rules without having to execute the same ones in an independent form. Next, some of them are described.

Composition without duplication of operations:

It integrates the behavior of two non-generic classes eliminating the duplication of operations, since a single class is obtained which gathers the whole behavior without duplication of those operations that were equivalent. The sequence of rules defining this strategy is: Factoring ➔ Composition ➔ Folding.

Factoring and joint of derived classes:

It factors equivalent operations in an existent base class and integrates the behavior of two non-generic classes that derive from that base class. The sequence of rules which defines this strategy is: Factoring ➔ Composition ➔ Folding.

Factoring of equivalent operations into a base class:

It is useful in those cases where the existence of equivalent operations is identified in derived classes and it is wanted to abstract such a behavior to the existent ancestor class. It reduces the duplication of operations in the derived classes and integrates the common behavior to the ancestor class. The sequence of rules is: Factoring ➔ Folding.

Abstraction of operations into a base class:

It promotes behavior to the ancestor class starting from a subset of operations without creating new classes and without increasing the amounts of hierarchy levels. The sequence of rules is: Abstraction ➔ Folding.

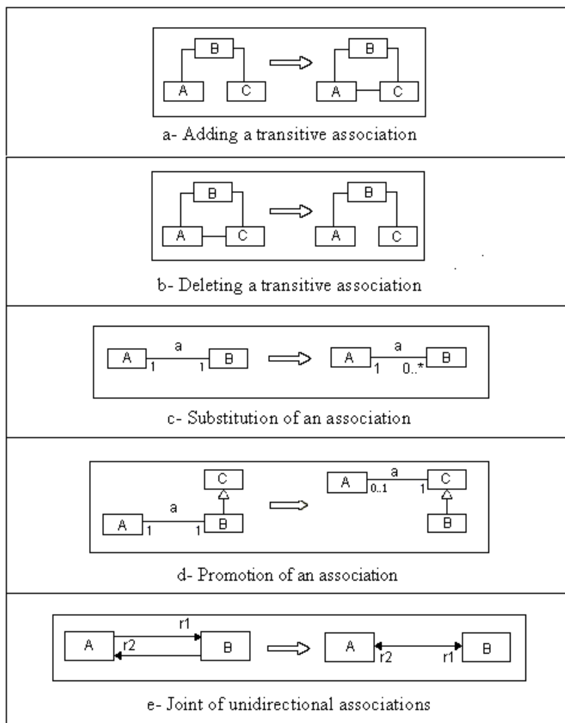
Unfolding by attributes into a base class:

It promotes behavior to the ancestor class starting from a subset A of attributes, without creating new classes and without increasing the amounts of hierarchy levels. The sequence of rules is: Unfolding (A) ➔ Folding.

3.2.3. Refactorings: Dealing with Associations

Some transformation rules dealing with associations are described below and they are shown in Figure 2.

Figure 2. Refactoring Associations



a- Adding a transitive association

b- Deleting a transitive association

c- Substitution of an association

d- Promotion of an association

e- Joint of unidirectional associations

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)

Deleting a transitive association:

Given an association between classes A and B, an association between classes B and C, and an association between A and C, the transitive association between A and C may be deleted (Whittle, 2002) .

Substitution of an association:

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

Promotion of an association:

Given an association R with multiplicity *mult1* (connected to a class A) and multiplicity *mult2* (connected to a class B), if B is a subclass, then R may be 'promoted' to the superclass of B with the condition that its multiplicity with A after the transformation is optional, i.e. 0∈ *mult1*. (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).

3.3. Defining rules

Transformation rules are denoted by

$$\xrightarrow[O]{I}[\ C$$

with input scheme I, output scheme O and applicability condition C.

All schemes in transformation rules are supposed to be syntactically valid and context correct. Within applicability conditions, syntactic and semantic ones can be distinguished.

As an example, the factorization rule definition is shown.

Input

- class (A, P_A, A_A, M_A, I_A, AS_A, H_A)
- class (B, P_B, A_B, M_B, I_B, AS_B, H_B)
- ***SetA*** = {(a1,a2) *where* a1∈ A ∧ a2∈ B ∧ a1.T = a2.T ∧ a1.A = a2.A ∧ *equiv-attr*(a1,a2)}
- ***M*** = {(m,n) / m ∈ M_A ∧ n ∈ M_B ∧ *equiv-oper*(m,n) }

Applicability Conditions

- Let A and B be classes without inheritance relationship between them.
- Each parameter of class A corresponds with an only parameter of class B and vice versa, i.e., PA = PB.
- A subset of attributes of A is related in a one-to-one way to a subset of attributes of B since its behavior is the same in both classes. *SetA* = {(a1,a2) where a1∈ A ∧ a2∈ B ∧ a1.T = a2.T ∧ a1.A = a2.A ∧ *equiv-attr* (a1,a2)}
- M ≠ Φ

Output

- class (C, P_C, A_C, M_C, I_C, AS_C, H_C) where:
  $P_C = P_A$,
  $H_C = H_A \cup H_B$.
  $\forall (m_i, n_j)$ M: *equiv-oper*(m_i , n_j) $\Rightarrow m_i \in M_C \wedge$
  ($\forall m_2 \in$ *set-oper* (m_i), $m_2 \in M_C$) ∧
  ∀a ∈ *set-attr*(m_i), a ∈ A_C ∧ (a.A = protected ∧ a.A = public)
  ∧ ∀ as ∈ *set-assoc*(m_i), as ∈ AS_C)
  that is to say, given an operation m_i ε M_A and another operation n_j ε M_B equivalent to the first one:

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

1. Move m_i up to the set of operations of the class C together with all operations referenced directly or indirectly in m_i.
2. Move all attributes referenced directly or indirectly in m_i up to the set of attributes of C and modify their access (public or protected)
3. Move all associations referenced in m_i up to the set of associations of C.

- class (A', $P_{A'}$, $A_{A'}$, $M_{A'}$, $I_{A'}$, $AS_{A'}$, $H_{A'}$) where:
  $P_{A'} = P_A$
  $A_{A'} = A_A - A_C$
  $M_{A'} = \{m / m \in M_A \wedge m \notin M_C\}$
  $I_{A'} = I_A$
  $AS_{A'} = AS_A - AS_C$
  $H_{A'} = \{(C, \text{public}, P_C)\}$

If $A_{A'} = \emptyset \wedge M_{A'} = \emptyset$ then the rule does not generate class A'. The inheritance of the direct descendants of A' is modified, i.e., the descendants of A' will become direct descendants of C.

- class (B', $P_{B'}$, $A_{B'}$, $M_{B'}$, $I_{B'}$, $AS_{B'}$, $H_{B'}$) where:
  $P_{B'} = P_B$
  $A_{B'} = A_B - A_X$, where A_X is the set of attributes of B that correspond to attributes of A which were factored to C, i.e.: $A_X = \{v / v \in A_B \wedge \exists \text{ parA} = (v1, v) \in \text{SetA such that } v1 \in A_C\}$
  $M_{B'} = \{n / n \in M_B \wedge \neg\exists\, m \in M_C$: *equiv-oper*$(m,n)\}$, i.e., the operations of class B which did not have an equivalent operation in class A which has been factored to the new class C.
  $I_{B'} = I_B$
  $AS_{B'} = AS_B - AS_C$
  $H_{B'} = \{(C, \text{public}, P_C)\}$

If $A_{B'} = \emptyset \wedge M_{B'} = \emptyset$ then the rule does not generate class B'. The inheritance of the direct descendants of B' is modified, i.e., the descendants of B' will become direct descendants of C.

4- AN EXAMPLE

The class diagram given in Figure 3.a. is a simple model of a graphical hierarchy for a means of transport. There is an associations between *Bus* and *Engine* and another between *Truck* and *Engine*.

The first step of the refactoring process consists in detecting possible restructurings. First, a set of operations M, with equivalent behavior is detected:

M = {(stop, stop), (brake, brake), (speedUp, speedUp)}

Next, a set of attributes SetA of both classes which correspond with each other is also detected:

SetA = {(speed, speed), (maxSpeed, maxSpeed)}

The requirements for the application of the rule of class refactoring are established. The necessary transformations are applied if the designer approves it. Most of the transformations are carried out in an automatic way; however, the designer intervention is sometimes required.

The application of the factoring rule leads to:

- Create a base class
- Rename the generated base class as *Highway* (given by the designer)
- Make the classes *Bus* and *Truck* subclasses of *Highway*
- Move operations m_i Î M up to *Highway*
- Move attributes a_i Î SetA up to *Highway*
- Replace the associations *Truck-Engine* and *Bus-Engine* by an association *Highway-Engine*, since the associations are referenced in the factored operations and they have the same characteristics.

Figure 3. Refactoring Class Diagram

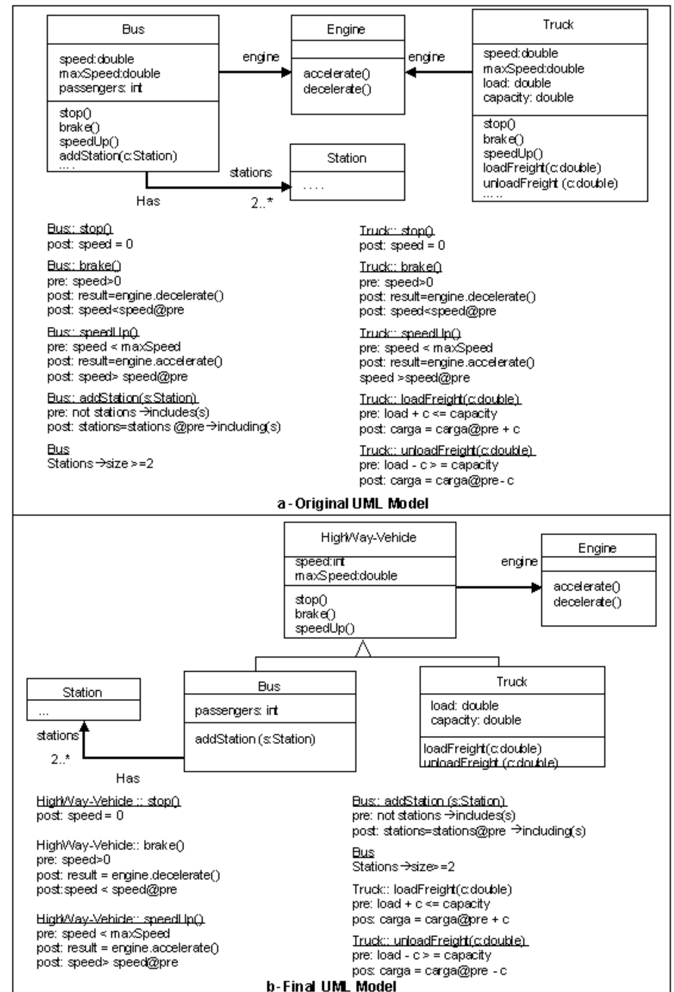


Figure 3.b. shows the resulting diagram. The transformation system allowed detecting two classes, originally not related, which are variations from the same general notion and may be moved up into a common ancestor. The resulting model shows a more advantageous design due to the elimination of duplicated attributes, operations and associations.

5. EXPERIMENTAL REFACTORING PROTOTYPE

To demonstrate the feasibility of this approach, a prototype which assists in the refactoring on object oriented hierarchies in C++ was implemented.

The prototype implements a small, rather powerful set of basic transformation rules (folding, unfolding, abstraction, composition, factorization) and the strategies presented in section 3.2.2.

In this approach, mechanical tasks are performing model transformations, verifying conditions of transformation rules and keeping track of the development process. On the other hand, typical creative aspects are the selection of rules and strategies. The prototype could store the history of the refactoring. This history serves as a detailed documentation and is the key aid to traceability. Developments are recorded automatically and could be replayed in order to accommodate changes.

The set of transformation rules was developed in Mathematica. A detailed description may be found in (Enriques, 2002).

6- CONCLUSIONS

This work presents a rigorous "rules + strategies" approach for the refactoring on UML static models. Our focus is on behavior-preserving model-to-model transformations. Transitions between versions are made according to precise rules based on the redistribution of classes,

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

variables, operations and associations across the diagram in order to facilitate future adaptations and extensions. Most of the transformations can be undone, which provide traceability in model refactoring.

A formal semantics was developed for a subset of UML metamodel which can be used to control the impact of refactorings.

Although the set of rules allows doing quite a number of interesting refactorings, it is limited since it does not focus on transformations which involve different UML views. Also, a number of extension rules associated to design patterns are currently investigated.

An experimental tool prototype to restructure object oriented hierarchies was described. It could be refined to be a practical tool for refactoring.

REFERENCES

Beck, K. (2000) *Extreme Programming explained.* Addison-Wesley.

Demeyer, S., Du Bois, B., Stenten, H. & Van Gorp, P. (2002). *Refactoring: Current Reasearch and Future Trends Language Description*, Tools and Applications. (LDTA 2002).

Enriques, S., Mariezcurrena, C. & Ortega, M. (2002). *Object Oriented Hierarchies Refactoring*. Undergraduate thesis, TR 287. Universidad Nacional del Centro de la Provincia de Buenos Aires, Argentina.

Evans, A. (1998). Reasoning with UML Class Diagrams. *Workshop on Industrial Strength Formal Method, IEEE Press.*

Fanta, R. & Rajlich, V. (1998). Reengineering an Object Oriented Code. *Proc. of IEEE International Conference on Software Maintenance*, 238-246.

Fanta, R. & Rajlich, V. (1999). Reestructuring Legacy C Code into C++. *Proc. of IEEE International Conference on Software Maintenance*, 77-85.

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

Gogolla, M.& Richters, M. (1998). Transformation Rules for UML Class Diagrams. *Proc UML' 98 Workshop*, Springer-Verlag, Berlin, 92-106.

Kollmann, R. & Gogolla, M. (2001). Application of UML Associations and Their Adornments in Design Recovery. *8th Working Conference on Reverse Engineering, IEEE*, Los Alamitos.

Moore, I. (1995). Guru – A tool for Automatic Restructuring of Self Inheritance Hierarchies. *Tools USA 95 (Technology of Object-Oriented languages and Systems, Tools 17)*, 267-275.

OMG (2003) *Unified Modeling language Specification*, v. 1.5. Object Management Group. Available at www.omg.org.

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

Philipps, J. & Rumpe, B. (2001). Roots of refactoring. *Proc. 10th OOPSLA Workshop on Behavioral Semantics*, Tampa Bay, Florida, USA.

Porres, I. (2003). Model Refactorings as Rule-Based Update Transformations. *Proc. of <<UML 2003>>*, Springer Verlag. Available at www.tucs.fi/Publications

Roberts, D., Brant, J. & Johnson, R. (1997). A refactoring tool for Smalltalk, *Theory and Practice of Object Systems,* Vol 3, N° 4.

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

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

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

TogetherSoft, ControlCenter (2003). Available at www.togethersoft.com

Van Gorp, P., Stenten, H., Mens, T. & Demeyer, S. (2003). Towards automating source-consistent UML Refactorings. *Proc. of <<UML 2003>>*, Springer Verlag. Available at win-www.uia.ac.be/u/lore

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

0 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage:
www.igi-global.com/proceeding-paper/refactoring-uml-class-diagram/32412

Related Content

Self-Awareness and Motivation Contrasting ESL and NEET Using the SAVE System
Laura Vettraino, Valentina Castello, Marco Guspiniand Eleonora Guglielman (2018). *Encyclopedia of Information Science and Technology, Fourth Edition (pp. 1559-1568).*
www.irma-international.org/chapter/self-awareness-and-motivation-contrasting-esl-and-neet-using-the-save-system/183870

A Hybrid Approach to Diagnosis of Hepatic Tumors in Computed Tomography Images
Ahmed M. Anter, Mohamed Abu El Souod, Ahmad Taher Azarand Aboul Ella Hassanien (2014). *International Journal of Rough Sets and Data Analysis (pp. 31-48).*
www.irma-international.org/article/a-hybrid-approach-to-diagnosis-of-hepatic-tumors-in-computed-tomography-images/116045

Analyzing and Predicting Student Academic Achievement Using Data Mining Techniques
Eric P. Jiang (2015). *Encyclopedia of Information Science and Technology, Third Edition (pp. 2453-2461).*
www.irma-international.org/chapter/analyzing-and-predicting-student-academic-achievement-using-data-mining-techniques/112661

A Study of Contemporary System Performance Testing Framework
Alex Ngand Shiping Chen (2018). *Encyclopedia of Information Science and Technology, Fourth Edition (pp. 7563-7576).*
www.irma-international.org/chapter/a-study-of-contemporary-system-performance-testing-framework/184452

A Systematic Review on Author Identification Methods
Sunil Digamberrao Kaleand Rajesh Shardanand Prasad (2017). *International Journal of Rough Sets and Data Analysis (pp. 81-91).*
www.irma-international.org/article/a-systematic-review-on-author-identification-methods/178164