JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol.8 , No.6 , September-October 2009

Classification of model refactoring approaches

Maddeh Mohamed, LI3 Ecole Nationale des Sciences de l'Informatique
Campus Universitaire Manouba - Manouba 2010 Tunis, Tunisia

Mohamed Romdhani, INSAT Centre Urbain Nord

Khaled Ghedira, SOIE, Ecole Nationale des Sciences de l'Informatique
Campus Universitaire Manouba - Manouba 2010 Tunis, Tunisia

Abstract

In this paper, we provide a detailed overview of existing researches in the field of software restructuring and model refactoring, from a formal as well as a practical point of view. We propose a possible taxonomy for the classification of several existing and proposed model refactoring approaches. The taxonomy is described with a feature model that makes the different design choices for model refactoring explicit.

1 INTRODUCTION

Refactoring is a technique to improve the maintainability of software systems by changing the internal structure of software without altering its external behavioral properties [Opdyke92]. Especially when automated by a tool, refactoring is an easy, quick and safe way to improve software systems at the code level, and to assist to identify errors. In addition, lightweight development methods, such as eXtreme Programming (XP) [Succi01], have promoted refactoring as a core development practice.

Refactoring is used to improve the quality of the software (e.g., extensibility, modularity, reusability, complexity, maintainability...). It is also used in the context of reengineering [Demeyer02], refactoring is needed to convert legacy code or deteriorated code into a more modular or structured form, or even to migrate code to a different programming language.

The majority of previous researches on refactoring focus on the code level and are less concerned with the earlier stages of design. A promising approach is to deal with refactoring in language independent way. It offers a solution to the reuse possibilities in the development of the refactoring primitives when they are adapted to new languages.

Considering the model-driven engineering [MDA06] which is an approach of software engineering where the primary focus is on models, as opposed to source code.

Model transformation [RFP02] is considered to be the heart of model-driven engineering. We can apply the refactoring as a model transformation. We transform an input model needing a design improvement to a target model using behavior-preserving transformations. There exist several classifications of model transformations like classification in “exomorphic”, “endomorphic” and “creational” transformations [Czarnecki03]. As presented in figure 1, we are interested only on endomorphic transformation. It deals with models represented at the same level of abstraction, and where source and target models are instances of the same metamodel. Usages of endomorphic transformations are numerous. Typical example of this kind of transformation is refactoring.

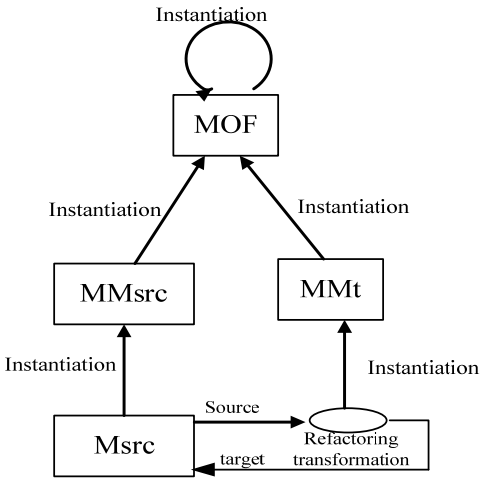


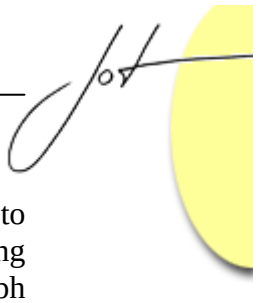
Figure 1: Model refactoring process

The remainder of the paper is structured as follows. Section 2 presents the code refactoring researches. Section 3 presents the model refactoring states of the art. Section 4 gives the model refactoring taxonomy. Section 5 discusses the different classification features. Section 6 and 7 are reserved for related works and conclusion, respectively.

2 CODE REFACTORING : STATE OF THE ART

This section presents the most relevant works on code refactoring , we try to give an indicative list of such works.

William Opdyke’s PhD thesis [Opdyke92] was the first major written work using the refactoring term. He considered refactorings for object-oriented software. For these refactorings, he described the design prerequisites and automatic program restructurings required to guarantee preservation of behavior. An important consequence of Opdyke’s work was the later development of a refactoring tool for Smalltalk. Roberts [Roberts99] extended Opdyke’s work by providing a more formal basis for composing refactorings and examined the use of dynamic information in refactoring.



Graph transformations [Corradini02] [Engels96] [Ehrig00] represents another way to deal with restructuring: the software is represented as a graph, and restructuring corresponds to the transformation rules .In [Jahnke97], authors suggest the use of graph transformation in order to replace occurrences of poor design patterns in a legacy program by good design patterns. In [Lakhotia98], the author present a transformation called tuck for restructuring programs by decomposing large functions into small functions. Tuck consists of three steps: Wedge, Split, and Fold. A wedge (i.e. a subset of statements in a slice) contains computations that are related and that may create a meaningful function. The statements in a wedge are split from the rest of the code and folded into a new function.

In [Snelting98], Snelting et al. present a framework for detecting and remediating the imperfect design of a class hierarchy based on concept analysis. Authors analyzes the class hierarchy along with a set of applications that use it, and constructs a lattice that provides valuable insights into the usage of the class hierarchy in a specific context and then generate a restructured class from the lattice.

In the works of Marticorena et al. [Marticorena06], we find another way to deal with refactoring which is software metrics. At first, authors propose a set of additional criteria to classify bad smells (define in an informal way code flaws, in order to suggest refactorings). Then they link the concept of bad smells with the concept of metric features. The aim of this work is to propose a method to evaluate the suitability of the tools assisting bad code smell detection, as well as the selection and the implementation of metrics linked with bad code smells.

In [Melton06], H. Melton and E. Tempero present a tool (Jepends) that analyses the source code of a system in order to identify classes as possible refactoring candidates. The tool analysis is based on the identification of the dependency cycles among classes. After that authors show how dependency cycles detected by the Jepends tool can be used as the starting point for refactoring.

Another refactoring approach is proposed in [Hadar06]. The Composition Refactoring Triangle (CRT) unified approach for handling multiple changes across complex environments. The CRT is a combination of three pillars: the CR Process (CRP); the CR Management Tool (CRMT); and the code complexity management procedure and version control, using External and Internal Composition Refactoring (ECR/ICR) XML markers. The external ones are linked to external visible features or software requirement changes, and the Internals are a set of uniquely defined refactoring primitives. Authors propose a practical method for the resolution of the problem, evaluating and estimating the time and effort required for the refactoring. The approach is assisted by techniques and tools, allowing the development team in handling multiple changes using combination of primitive refactorings.

3 MODEL REFACTORING STATE OF THE ART

A recent trend is to apply the concepts of refactoring to higher levels of abstraction. Consequently, model refactoring is emerging as a desirable means to improve models design using behavior-preserving transformations. Although it exists many researches on code refactoring, works on model refactoring are still young. This section presents the most relevant researches on model refactoring.

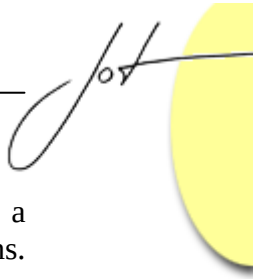
In [Biermann06], Enrico Biermann et al. propose to use the Eclipse Modeling Framework (EMF), a modeling and code generation framework for Eclipse applications based on structured data models. They introduce the EMF model refactoring by defining a transformation rules applied on EMF models. EMF transformation rules can be translated to corresponding graph transformation rules. If the resulting EMF model is consistent, the corresponding result graph is equivalent and can be used for validating EMF model refactoring. Authors offer a help for developer to decide which refactoring is most suitable for a given model and why, by analyzing the conflicts and dependencies of refactorings. This demarche is closed to the model driven architecture (MDA) paradigm [MDA06] since it starts from the EMF metamodel applying a transformation rules.

Another work on model refactoring is proposed in [Zhang05], based on the Constraint-Specification Aspect Weaver (C-SAW)1, a model transformation engine which describes the binding and parameterization of strategies to specific entities in a model. Authors propose a model refactoring browser within the model transformation engine to enable the automation and customization of various refactoring methods for either generic models or domain-specific models. The transformation proposed in this work is not based on any metamodel, it is not an MDA approach.

In [Rui03] Rui, K. and Butler, apply refactoring on use case models, they propose a generic refactoring based on use case metamodel. This metamodel allows creating several categories of use case refactorings, they extend the code refactoring to define a set of use case refactorings primitive. This refactoring is very specific since it is focused only on use case model, the issue of generic refactoring is not addressed, and these works do not follow the MDA approach.

R. Marticorena affirms that on one side, all tools such as (IntelliJ IDEA, Eclipse, Refactoring Browser, JRefactory...) approach the implementation and execution of refactorings from the scratch, with a solution based on customized libraries, not supporting reuse to compose and run refactorings on other languages with similar features. On the other side, the modern software systems often require different modules developed in different languages. As solution, he proposes in [Marticorena05] an independent language refactoring based on MOON [Marticorena03] a minimal notation to support most of the abstract concepts included in a big family of strongly typed object-oriented languages. Author analyze and define a refactoring catalog based on MOON, he try to obtain a formal support to the definition of refactoring that can be achieved with

[1] Additional information about C-SAW is available at: http://www.cis.uab.edu/gray/Research/C-SAW.



language independence preserving the behavior of the software. This work proposes a measure the software quality improvement, resulting from the refactoring operations. This approach is very closed to the MDA paradigm since they start from the MOON as the equivalent of the MOF, but MOON is not a standard, as matter of fact, it cannot offer a generic and extensible approach. Using MOON we are faced to an interoperability problem when moving to another meta-metamodel.

In the line of language independent refactoring and metamodelling, Sander et al. [Tichelaar00], study the similarities between refactorings for Smalltalk and Java, and build the FAMIX model. It provides a language-independent representation of object-oriented source code. It is an entity-relationship model that models object-oriented source code at the program entity level, with a tool to assist refactoring named MOOSE. FAMIX [Tichelaar99a] [Tichelaar99b] does not take account neither complex features in strongly typed languages (this point have been discussed in [Marticorena05]), nor aspects of advanced inheritance and genericity. This approach is not really independent from language since the refactoring transformation is achieved directly on the original code. This alternative forces to implement transformers of specific code for each language. These code transformers use an approach based on text using regular expressions.

Based on their experience of the FAMIX metamodel Pieter Van Gorp et al. propose in [Gorp03] an extension of the UML metamodel for resolving inconsistency problems that arise when performing a model refactoring. This inconsistency occurs between a design model and the corresponding code. Typical MDA tools using the UML metamodel consider the whole method body as implementation specific, and when performing a model refactoring, we do not pay attention to the implementation level to increase the abstraction level. Consider the simple Rename Class refactoring primitive, at implementation level the type casts and exceptions, will not be updated accordingly to the new name. The solution proposed in [Gorp03] is an extension of the UML metamodel for refactoring with aims of relating a method body to its contained statements and then leverage the profile mechanism to model language specific features such as conditionals, exceptions, type casts... as cross-language abstractions that express strictly the information needed by the refactoring catalog in use. As result, they propose the GrammyUML metamodel, which reuse UML 1.4 action entity, it contains arguments and introduces the SingleTargetAction relating the action to its target. Finally they apply an UML profile for refactoring developed on top of the GrammyUML metamodel. This approach is not closed to the MDA paradigm, it does not define a meta-metamodel level and there is no evaluation of the extension proposed with the MOF or any other meta-metamodel.

Another model refactoring is presented in [Kempen05], based on SAAT (Software Architecture Analysis Tool). It allows calculating metrics about UML models the metrics are then used to identify the flaws or anti-patterns. Authors represent the structure using class diagrams, and the behaviour of each class using statecharts. After that they examine the metrics for refactoring a centralized control structure into one that employs more delegation they use csp-based formalism (Communicating Sequential Processes) to

guarantee the behaviour preservation. This work deal with a particular problem of centralized control structure, it is not based on any metamodel.

In the next section other works on model refactoring are referenced.

4 DESIGN FEATURES OF MODEL REFACTORING APPROACHES

Based on the state of the art defined in the section 3 we identified various properties of the model refactoring domain. Indeed, we elaborated a manual classification which puts back the main characteristics presented in a unified way. This presentation allows to highlight the common points between each approach and to consider a variety of potential combinations over them. Czarnecki et al, proposes a possible taxonomy for the classification of several existing and proposed model transformation approaches. Considering the refactoring as a model transformation problem, we extend the feature diagrams resulting from a domain analysis of existing model transformation approaches presented in [Czarnecki03].

The taxonomy for the classification of model transformation is valid for model refactoring but not sufficient. We enlarge this taxonomy by adding new concepts, specific to the model refactoring domain.

Every concept is then detailed and followed by a comparative table relating the presence of the concept in the works presented above.

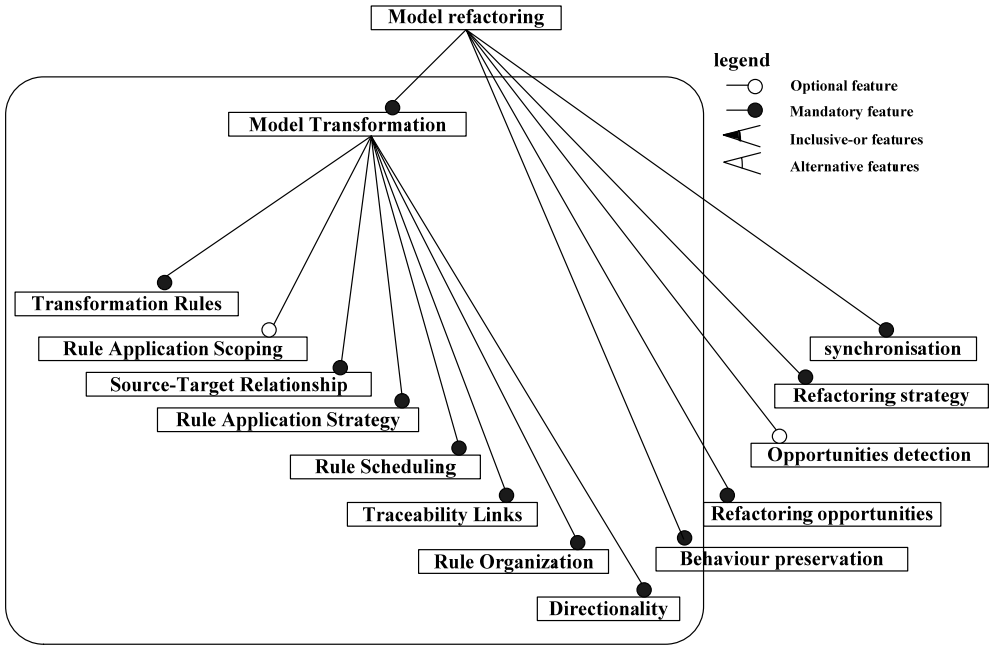


Figure 2: Features diagram for model refactoring

Figure 2 presents the model refactoring taxonomy. The part in box represents the model transformation taxonomy proposed by Czarnecki et al [Czarnecki03]. We add the new features related to the model refactoring domain.

Behaviour preservation

A model refactoring must preserve the observable behaviour of the model it is transforming. In order to achieve this, we need a precise definition of "behaviour" in general, and for models in particular.

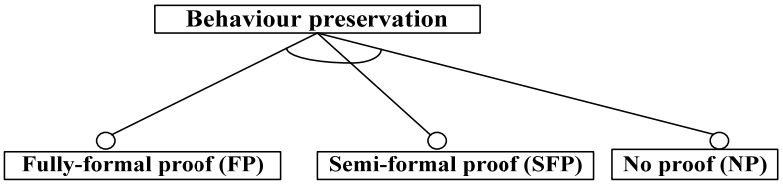


Figure 3: Features of behaviour preservation

As presented in figure 3, existing refactoring works has generally relied on either a semi-formal demonstration of behaviour preservation (e.g. approach based on contracts : pre, postconditions and invariants [Marticorena05]), or indeed no demonstration of behaviour preservation at all [Biermann06] [Zhang05]. In practice, full behaviour preservation is very difficult to prove. It may be some tolerance for changing behaviour as long as we are able to identify precisely how a given model transformation modifies it.

Behaviour preservation	(Sander et al., 2000)	(Rui et al., 2003)	(Pieter et al., 2003)	(Slavisa., 2004)	(Zhang et al., 2005)	(Raul et al., 2005)	(Marc et al., 2005)	(Ragnhild et al., 2006)	(E. Biermannet al., 2006)	(Moha et al., 2008)
FP							×			
SFP	×		×	×		×			×	
NP		×		×	×			×		×

Table 1: The comparison of the various approaches with regard to the properties of behaviour preservation

Refactoring opportunities

The figure 4 presents the refactoring opportunities which contain the possibilities of detection of the design defects and the possible improvements, this property is crucial for the creation of automatic refactorisations

The refactoring strategy includes the detection whether the refactoring needs in a source model. Users can specify the refactoring needs, as response for specific changes. This is a reactive approach, the programmer detects the suitable refactoring (e.g. the work presented in [Tichelaar00] do not indicate where apply a refactoring it only explain how apply a model refactoring primitive). It can also be detected automatically from a

proactive approach (inference), in this case, the system detects the refactoring opportunities by analyzing the design defects.

Design defects are poor design choices that hinder the maintenance of programs. They include bad solutions to recurring problems in object oriented design, such as antipatterns [Brown98] (as opposed to design patterns [Larman02]), defects related to design and code smells [Fowler99] (symptoms of design defects).

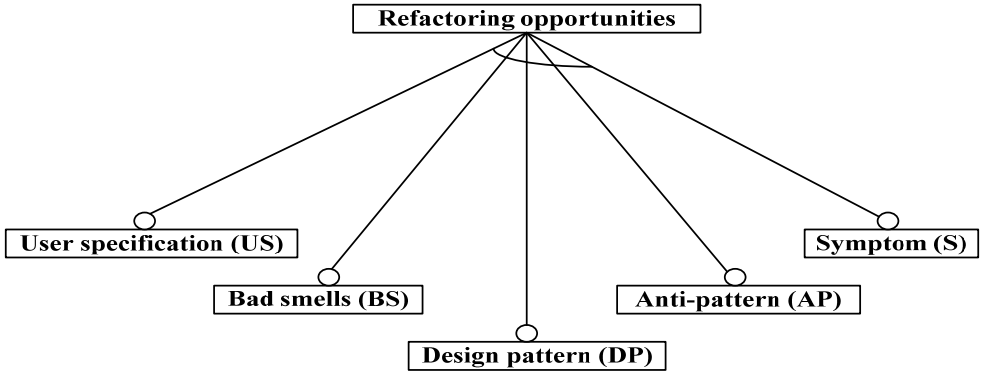


Figure 4: Features of refactoring opportunities

Refactoring opportunities can be identified by analyzing bad smells (e.g. in [Marticorena05] authors apply a refactoring inference based on metric values and bad smells) or antipatterns (e.g. in [Kempen05] metrics are used to identify the anti-patterns). Symptoms serve to detect the model elements susceptible to be refactored, as well as bad smells. It exist other solution not covered by this classification. As example, in [Moha08], authors propose an automated approach for suggesting defect-correcting refactorings using relational concept analysis (RCA). In [Moha06], authors combine the effectiveness of metrics with formal concept analysis to detect design defect. This approach is not independent from the language and therefore it can not be generic.

		(Sander et al., 2000)	(Rui et al., 2003)	(Pieter et al., 2003)	(Slavisa., 2004)	(Zhang et al., 2005)	(Raul et al., 2005)	(Marc et al., 2005)	(Ragnhild et al., 2006)	(E. Biermannet al., 2006)	(Moha et al., 2008)
Refactoring opportunities	US	×	×	×	×	×	×	×	×	×	×
	BS						×				×
	DP							×			×
	AP							×			×
	S						×		×		×

Table 2: The comparison of the various approaches with regard to the properties of refactoring opportunities

Opportunities detection

As presented in figure 5, in case when the detection of model refactoring opportunities is achieved automatically from a proactive approach, it then necessities a prerequisite analyze. We distinguish two analyzing ways, the first one is focused on code level and then applied on model (e.g. [Tichelaar00] [Moha08] [Moha07]) , the second one perform analysis on model (e.g. [Kempen05] [Straeten06] [Straeten03]).

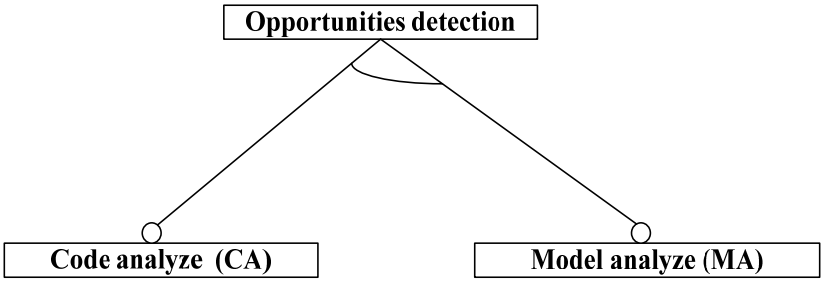


Figure 5: Features of opportunities detection

This approach is more difficult since model level contains less information to analyze than the code level, although it has the advantage to offer a generic and reusable refactoring independent from any platform.

		(Sander et al., 2000)	(Rui et al., 2003)	(Pieter et al., 2003)	(Slavisa., 2004)	(Zhang et al., 2005)	(Raul et al., 2005)	(Marc et al., 2005)	(Ragnhild et al., 2006)	(E. Biermannet al., 2006)	(Moha et al., 2008)
Opportunities detection	CA	×									×
	MA		×	×	×	×	×	×	×	×	

Table 3: The comparison of the various approaches with regard to the properties of opportunities detection

Synchronization

The Model refactoring operation must guarantee the consistency between a design model and the corresponding code.

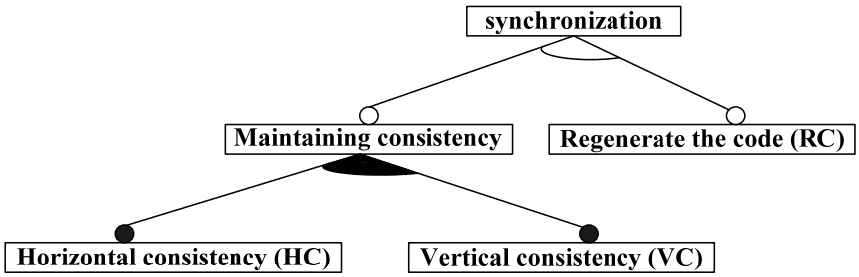


Figure 6: Features of synchronization

We decompose in figure 6 the maintaining model consistency feature in two orthogonal aspects: horizontal consistency must be maintained between design diagrams like UML static and dynamic diagrams (e.g. [Straeten06] [Straeten03]) and vertical consistency between model and code. Model transformation is based on the metamodel level, and the whole method body is considered in the MDA paradigm as implementation specific related to the code level. Therefore, when model is refactored it can cause an inconsistency problem. Consider the simple Rename Class refactoring: class names may be used within protected areas like in type declarations, type casts and exceptions, and the new name will not be updated accordingly [Gorp03], since this information are not stored in model level.

Face to this problem we can adopt two strategies, the first one is to try to relate the model to its code and then apply the profile mechanism to model the language specific features such as conditionals, exceptions, and type casts, (e.g. [Gorp03]). Her we have a compromise between maintaining consistency and the reusability, since relating model to code offer a specific solution not applicable to other platforms.

The second one is related to the MDA process which starts from model arriving to the code by automated transformations. Performing a model refactoring is done at Platform Independent Model (PIM) or Platform Specific Model (PSM) levels and then code will be regenerated reflecting the new features on the refactored model. This approach is based on both model-to-model and model-to-code transformations and offer a generic and reusable refactoring independent of any platform.

		(Sander et al., 2000)	(Rui et al., 2003)	(Pieter et al., 2003)	(Slavisa., 2004)	(Zhang et al., 2005)	(Raul et al., 2005)	(Marc et al., 2005)	(Ragnhild et al., 2006)	(E. Biermannet al., 2006)	(Moha et al., 2008)
synchronization	RC	×	×		×	×		×	×	×	
	HC								×		
	VC			×			×				×

Table 4: The comparison of the various approaches with regard to the properties of refactoring strategy

Refactoring strategy

As presented in figure 7, although model refactoring can be done manually using any graphical tool. It is possible to do that for small applications, but when we move to large systems this operation will be hardest and time consuming, needing effort and experience. Works presented in this paper try to resolve this problem by offering an automated and a semi automated refactoring.

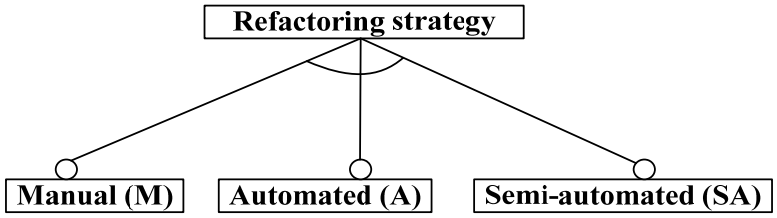


Figure 7: Features of refactoring strategy

Automated refactoring, deal with the automated detection and correction of design defects. The programmer has just to execute the refactoring (e.g. [Straeten06]). Semi-automated refactoring relay on the interaction with programmer. We can propose a list of model refactoring primitives, and user can apply any primitives depending on his preferences (e.g. [Sunye01]). Markovic in [Markovic04] presents a set of refactoring rules that can be checked, reused and composed. Author offer an algorithm to compute the description of sequentially composed transformations allowing users to check if a sequence of transformations is successfully applicable for a given model.

		(Sander et al., 2000)	(Rui et al., 2003)	(Pieter et al., 2003)	(Slavisa., 2004)	(Zhang et al., 2005)	(Raul et al., 2005)	(Marc et al., 2005)	(Ragnhild et al., 2006)	(E. Biermannet al., 2006)	(Moha et al., 2008)
Refactoring strategy	M		×		×						
	A						×				×
	SA	×		×		×	×	×	×	×	

Table 5: The comparison of the various approaches with regard to the properties of refactoring strategy

5 DISCUSSION

Although there are satisfactory solutions for the code refactoring, (such as IntelliJ IDEA, Eclipse, Refactoring Browser, JRefactory), the models refactoring still in the stage of research and development. In this context, several propositions emerge, certain try to supply a structured demarche, the others propose ad-hoc solutions. Our study aims to classify these various approaches to be able to compare them, possibly to combine them she also allows to identify future challenges. According to the analysis of the various tables presented in the section 3 we notice that for the properties of:

- The behaviour preservation: most of the works do not offer formal proofs to validate the refactored model and guarantee that it offers the same behavior of the source model. Some works are not interested at all to this property although it is fundamental.
- Refactoring Opportunities: although there are several works which allow identifying the possible design defects, most of the approaches do not reuse the experiences of the designers such as the design patterns. An interesting way to be

investigated would be to try to integrate these experiences for the automatic detection of the refactoring opportunities.

- Synchronization: the various approaches proposed in the literature ignore the horizontal aspect of the synchronization although its importance to maintain the coherence between the various static and dynamic diagrams. Some works try to maintain a consistency between the model and the code. Validating these approaches and generalizing them still a challenge.
- Refactoring Strategy: to propose a totally automatic refactoring does not have to exclude the intervention of the designer. Most of the approaches propose a semi-automatic and manual solution. We suggest integrating the approaches of refactoring opportunities detection to offer automatic solutions in a complete framework.

6 RELATED WORK

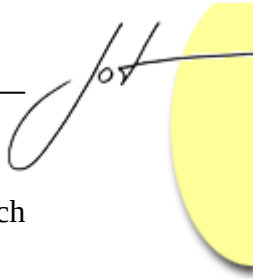
Czarnecki et al. propose in [Czarnecki03] a classification of model transformation approaches. This classification is applicable for model refactoring since model refactoring is an endomorphic model transformation. Although, model refactoring introduces new concepts not covered in such taxonomy. This taxonomy is based on a set of features identified from the model transformation domain analysis.

Tom Mens et al. offer in [Mens05] another taxonomy of model transformation based on the discussions of a working group on model transformation of the Dagstuhl Seminar on Language Engineering for Model Driven Software Development. The two works are relatively similar since we find the same concepts. Mens et al. focuses on helping the developer choosing a particular transformation language by answering crucial questions for model transformation they proposes. In [Mens03] Tom Mens propose a list of the most important challenges in model refactoring that could be investigated.

Although, this works surround the model refactoring area, it is necessary to define a well established classification of model refactoring approaches this paper provide a set of classes in which we can catalogue each work dealing with model refactoring.

7 CONCLUSION

Model refactoring is a young area, although it is related to and builds upon the more established fields of program transformation and meta-programming. Many approaches to model refactoring have been proposed over the last years, but little experience is available to assess their effectiveness in practical applications. In this respect, we are still at the stage of exploring possibilities and eliciting requirements. In this paper we presented the most relevant works on model refactoring based on our intuition and the application examples published together with each approach. We classified the existing model refactoring approaches relating to refactoring features we proposed. A depth



comparison based on benchmark problems would be the next step to evaluate each approach.

REFERENCES

[Corradini02] Corradini A., H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors,"Graph Transformation" *Lecture Notes in Computer Science 2505*, Springer-Verlag, 2002.

[Czarnecki03] Czarnecki K., Helsen S. "Classification of Model Transformation Approaches" . *In online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. Anaheim*, October, 2003.

[Roberts99] Donald Roberts. "Eliminating Analysis in Refactoring". *PhD dissertation*, University of Illinois at Urbana-Champaign, Department of Computer Science, 1999.

[Engels96] Engels G., E. Hartmut and G. Rozenberg, editors, "Special Issue on Graph Transformations" *Fundamenta Informaticae 26(3,4), IOS Press*, 1996.

[Biermann06] E. Biermann, K. Ehrig, G. Kuhns, C. Köhler, G. Taentzer, and E.Weiss."EMF Model Refactoring based on Graph Transformation Concepts", *Electronic communications of the east*, Volume 3, 2006.

[Ehrig00] Ehrig, H., G. Engels, H.-J. Kreowski and G. Rozenberg, editors, "Theory and Application to Graph Transformations" *Lecture Notes in Computer Science 1764*, Springer-Verlag, 2000.

[Hadar06] Ethan Hadar, Irit Hadar, "The Composition Refactoring Triangle (CRT) Practical Toolkit: From Spaghetti to Lasagna", *OOPSLA 2006*, Portland, Oregon, USA. ACM 1-59593-491-X/06/0010.

[Succi01] Giancarlo Succi, Michele Marchesi, "Extreme Programming Examined", Addison-Wesley, 2001.

[Sunye01] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel, "Refactoring UML Models", In *Proceedings of UML 2001*, Volume 2185. Springer Verlag, 2001.

[Larman02] Graig Larman, "UML et Design Patterns", 2002.

[Melton06] Hayden Melton, Ewan Tempero, "Identifying Refactoring Opportunities by Identifying Dependency Cycles", *Twenty-Ninth Australasian Computer Science Conference (ACSC2006)*, Hobart, Tasmania, Australia ,Vol. 48, January 2006.

[Jahnke97] Jahnke J. H. and A. Zündorf, "Rewriting poor design patterns by good design patterns", in: S. Demeyer and H. Gall, editors, *Proc. of ESEC/FSE '97*

Workshop on Object-Oriented Reengineering, Technical University of Vienna, 1997.

[Lakhotia98] Lakhotia A. and J.-C. Deprez, "Restructuring programs by tucking statements into functions", in: M. Harman and K. Gallagher, editors, *Special Issue on Program Slicing, Information and Software Technology 40*, Elsevier, 1998.

[Kempen05] Marc Van Kempen, Michel Chaudron, Derrick Kourie, Andrew Boake, "Towards Proving Preservation of Behaviour of Refactoring of UML Models", *in proceedings of SAICSIT 2005*, Pages 252.

[MDA06] *MDA Guide Version 1.0.1*, omg/2003-06-01, 12th June 2003. Accessed on Jan 2006.

[Fowler99] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999.

[Moha08] Naouel Moha, Amine Mohamed Rouane Hacene, Petko Valtchev, and Yann-Gael Gueheneuc "Refactorings of Design Defects using Relational Concept Analysis", *ICFCA*, 2008.

[Moha06] Naouel Moha, Jihene Rezgui, Yann-Gael Gueheneuc, Petko Valtchev, and Ghizlane El Boussaidi, "Using FCA to Suggest Refactorings to CorrectDesign Defects", *CLA*, 2006.

[Moha07] Naouel Moha, "Detection and Correction of Design Defects in Object-Oriented Designs", *OOPSLA*, October 2007.

[Gorp03] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. "Enabling and using the UML for model driven refactoring". *4th International Workshop on Object-Oriented Reengineering (WOOR)*, (Germany), July 21st, 2003. Technical Report 2003-07 of the University of Antwerp (Belgium), Department of Mathematics & Computer Science, 2003.

[Marticorena06] Raul Marticorena, Carlos Lopez, and Yania Crespo, " Extending a Taxonomy of Bad Code Smells with Metrics", *WOOR'06*, Nantes, 4th July, 2006.

[Marticorena05] Raul Marticorena, "Analysis and Definition of a Language Independent Refactoring Catalog", *17th Conference on Advanced Information Systems Engineering (CAiSE 05)*. Portugal., page 8, jun 2005.

[Marticorena03] Raul Marticorena and Yania Crespo. "Refactorizaciones de especializacion sobre el lenguaje modelo MOON". *Technical Report DI-2003-02*, Departamento de Informatica. Universidad de Valladolid, septiembre 2003.

[Straeten06] Ragnhild Van Der Straeten, Maja D'Hondt "Model Refactorings through RuleBased Inconsistency Resolution", *SAC'06 Dijon*, France 2006.

[Straeten03] Ragnhild Van Der Straeten, Jocelyn Simmonds "Detecting Inconsistencies between UML Models Using Description Logic", *International Workshop on Description Logics, Rome, Italy* September 5-7, 2003.

[RFP02] *Request for Proposal: MOF 2.0 Query /Views /Transformations RFP*, OMG Document: ad/2002-04-10.

[Rui03] Rui K. and Butler, G. (2003). "Refactoring Use Case Models : The Metamodel". In Proc. *Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. CRPIT,* 16. Oudshoorn, M.J., Ed. ACS. 301-308.

[Tichelaar00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, Oscar Nierstrasz. "A Meta-model for Language-Independent Refactoring", *published in the proceedings of ISPSE* 2000.

[Tichelaar99a] Sander Tichelaar, Serge Demeyer and Patrick Steyaert. Famix 2.0 – the *famoos information exchange model*. URL: http://www.iam.unibe.ch/ famoos/FAMIX/, 09 1999.

[Tichelaar99b] S. Tichelaar, *FAMIX Java language plug-in 1.0,* Technical, Report, University of Berne, September 1999.

[Demeyer02] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns,* Morgan Kaufmann and DPunkt, 2002.

[Markovic04] Slavisa Markovic, "Composition of UML Described Refactoring Rules, OCL and Model Driven Engineering", *Lisbon, Portugal,* October 12, 2004.

[Snelting98] Snelting G. and F. Tip, "Reengineering class hierarchies using concept analysis", *in: Proc. Foundations of Software Engineering (FSE-6), SIGSOFT Software Engineering Notes 1998.*

[Mens03] Tom Mens, *First international workshop on refactoring :* achievements, challenges, and effects, 2003.

[Mens05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp, "A Taxonomy of Model Transformations", *Dagstuhl Seminar Proceedings,* 2005.

[Opdyke92] William F. Opdyke. "Refactoring Object-Oriented Frameworks". *PhD dissertation, University of Illinois at Urbana-Champaign*, Department of Computer Science, 1992.

[Brown98] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. Anti Patterns: "Refactoring Software, Architectures, and Projects in Crisis". *John Wiley and Sons,* 1st edition, March 1998.

[Zhang05] Zhang J., Lin, Y. and Gray, J. (2004) "Generic and Domain-Specific Model Refactoring using a Model Transformation Engine", *Model-driven Software*

Development – Research and Practice in Software Engineering, accepted for publication in 2005.

About the author

Mohamed Maddeh (maddeh_mohamed@yahoo.com) PhD. Student in National Institute of Computer Sciences of Tunis (ENSI). Member of LI3. Member of Tunisian association of Artificial intelligence (ATIA).

Prof. Khaled GHEDIRA (khaled.ghedira@isg.rnu.tn) is Engineer ENSEEIHT and ENSIMAG, he was DR in the Institute of Computing and Artificial intelligence of Neuchâtel in Swiss and consultant at British Telecom. He was the director of the National Institute of Computer Sciences of Tunis (ENSI) and president of the ATIA. He is the founder and responsible of the Research Unity SOIE (Strategies of Optimization of the engineering of the Information and the knowledge) actually LI3. His research interests include the AI more particularly the Multi-agents systems and the Constraints Satisfaction Problems (CSP), the logistics problems and\or the industrial automation (supply chain, VRP), the optimization multi-criteria, the stochastic optimization and\or heuristics (RS, AG, RT, ACF etc.). He is member of several committees of diverse conferences and journals.

Dr. Mohamed ROMDHANI (Mohamed.Romdhani@insat.rnu.tn) is a Doctor in computer sciences at INPG (Polytechnic National institute of Grenoble, France). He is teaching within the National Institute of Applied sciences and Technologies of Tunis (INSAT) and responsible of the programs of computing science and in particularly Software Engineering.