Analysis and Definition of a Language Independent Refactoring Catalog

Raúl Marticorena

University of Burgos
Languages and Informatic Systems
rmartico@ubu.es

Abstract. Refactoring is an emergent trend in software evolution. There are activities and work lines not yet covered in this process. Assistance to refactoring with a certain language independence is one of them.
This work presents a proposal to the refactoring process definition based on a model language, with the aim of establishing a solid base to the development of software tools, which use refactoring with a language independence.
In particular, we pretend to refactor in the context of object-oriented languages, statically typed with advanced inheritance and genericity.

Key words: refactoring, language independence, model language, object oriented programming.

1 Context

A research trend in refactoring process definition, analysis and automation, is to reach a certain language independence [1, 2]. This aim is twofold. On one side, refactoring tools are software systems we want to reuse. Adapting to new source languages and/or new refactoring operations from previous efforts in the definition and implementation.

On the other side, the modern software systems often requires different modules developed in different languages. Therefore integrated development environments with multiple language support are demanded and this also applies to incorporated refactoring tools.

Refactoring definition in a language independent way offers a solution to the reuse possibilities in the development of the refactoring tools when they are adapted to new languages. With this approach, the effort of defining refactorings in a general way guarantees a recovery of the initial effort and its future application in new languages.

This work presents an overview of the current research in refactoring as well as the opened work lines. Concretely, we establish a doctoral work to obtain a solution to the problem of language independence, cited by [3].

This paper is organized as follows: Section 2 presents the current state of the art in refactoring software, from the beginnings until the present work lines. In Section 3 we explain the problem definition. Section 4 establishs the aims and goals, and in Section 5, we introduce the expected solution to the problem. Finally, in Section 6, we describe the current progress and the work to be done.

2 State of the Art

2.1 Refactoring

In his works [4–6] E.Casais presents the problem of the absence of methodologies and tools to modify source code. His works describe change operations on object-oriented software. The author proposes the definition of software transformation primitives in an informal manner. On the base of these low level operations defines high level transformations and restructurings (applying the Demeter's Law, transforming the inheritance hierarchies, etc). In order to perform the transformation, he proposes a solution based on an algorithm. The algorithm is described in a formal way, in which all actions have associated a set of conditions. The practical work of the algorithm application is achieved with EIFFEL classes.

In [7] W. Opdyke analyzes the problem of software maintenance, focusing on the modification tasks. From previous works on software evolution, schema evolution in object-oriented databases and class reuse, he presents refactoring as *"restructuring plans which support changes to an intermediate level"*. About the design problem and the maintenance complexity, he defends the automatic support to the software modification through tools (from design as well as code). He describes an approach to the automatic support and restructuring of object-oriented software.

A basic feature of the new concept of refactoring is that it must preserve the program behavior. Refactoring supports the changes in such a way that improves the later comprehension and evolution of the software system. With respect to *when* we must refactor, he suggests the necessity of rules that allow a certain inference. Respect to *where* we must refactor, he exposes the necessity to carry out at least one or more interactions with the designer or programmer.

In [8] M.Fowler, retakes the concept of refactoring. The author's definition is: *"Refactoring is to restructure software by applying a series of refactorings without changing its observable behavior."* His basic objective is to improve the software structure. He proposes the idea of guarantee the behaviour through unit testing. This work contains also a refactoring catalog. The main problem of the refactoring definitions in this catalog is its informal definition. The operations are described in a free way, like a list of stages to achieve on the source code. The exposition is similar to other books following a structure of cook books like [9], where description and solution of the problem is posed in an informal manner, using in this case examples in JAVA language.

Problem of automatic software transformation, preserving the behavior, is showed also in [10]. The basic aim is the formal definition of refactorings, easy to automate through a first-order logic. Reusing the proposal of [7] and his definition of low level operations, the author propose the definition of preconditions and postconditions in each refactoring. Behavior preservation is based on the postcondition verification. This work studies a dynamically typed language like SMALLTALK. The implementation of his work is collected in the *Refactoring Brower* tool[1].

2.2 Language Independence

In the line of language independence refactoring, in [1, 11] is defined the FAMIX model like a metamodel to storage information with the aim of integrate several development environments, together with a tool to assist refactoring named MOOSE [12]. Like starting point to define the model, it is centered in the study of two languages with different features like SMALLTALK and JAVA. It does not take account complex features in strongly typed languages, aim of our previous studies [13, 14], nor aspects of advanced inheritance and genericity. MOOSE adopts like technique the use of asserts [7, 10] using the information from metamodel to validate the preconditions of the refactorings. Based on the validation of the preconditions, they warrant the behavior preservation, without provide a support to undo the last changes of a refactoring when there are possible exceptions in the execution of a transformation.

By other side, the associated transformation of the refactoring are 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. In these works they propose a future line including abstract syntax trees (AST).

About the software transformation with certain independence language, there are works like [15] oriented to the generation of parsers with the aim to define new transformations on the code.

2.3 Refactoring Opportunities Detection

Other authors look for a language independence representation with object-oriented code. In [16], the authors work in this line using logic programming languages on meta language. The logic predicates are used to check and recover information from a meta language. In this case, the aim of the authors is the identification of “Bad Smells” [8]. From a database of logic facts which map the information in the code, they suggest refactorings to erase the “Bad Smells”. To define these refactorings, they [17] propose a serie of transformations on the elements that conform the code. With the aim of preserve the transformation correctness, the preconditions are verified, similar to the proposal of [7, 10]. Although their works suggest a possible language independence, their works and prototypes are based on SMALLTALK.

Other proposals, in the definition of models to describe object-oriented languages, even though different aims, use technics based on frameworks like OFL (Open Flexible Languages) [18], or based on a graph software representation, like [19] whose aim is to obtain metrics with language independence.

[1] Available in http://wiki.cs.uiuc.edu/RefactoringBrowser. Ĺast visit 2nd May 2005

Finally, there is a proposal more ambitious in [20] to define generic refactoring with certain independence of programming paradigm.

3 Problem Definition

The effort made in the refactoring definition is not being reused currently. Solutions proposed to refactor demand a particular analysis and implementation for each object-oriented programming language (i.e. C++ [7], Smalltalk [10], Java [8], etc.)

Current integrated development environments that include refactorings (i.e. IntelliJ IDEA, Eclipse, etc.) are a clear example. Other solutions are specific tools to refactor (i.e. Refactoring Browser, JRefactory, etc.) All these tools 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.

The problem comes from an absence of refactoring definition with certain independence language, exposing the common elements to several languages and allowing their implementation and reuse later. Particularly, this PhD thesis proposal is focused in the problem of the refactoring definition on a object-oriented language suite, statically typed, considering the complex features like advanced inheritance (multiple inheritance) and generic classes.

With this aim, we take the model language MOON[2]. It was defined as a minimal notation to support most of abstract concepts included in a big family of strongly typed object-oriented languages [2]. It includes concepts as types, parametric types, classes, generic classes, inheritance, etc. This model language was also defined with the goal of support the refactoring definitions obtaining a certain language independence. The aim is to analyze and define a refactoring catalog on MOON:

- *Identify under which situations (symptoms) must be initiated a refactoring.* From a reactive viewpoint (under demand), where the programmer detects the suitable refactoring or from a proactive approach (inference) where the systems detects and marks the convenience or necessity of refactoring.
- *Define the refactoring from a point of view of language independence.* All the elements that compose a refactoring must be common to a language family. This is guaranteed by defining refactoring from the available information in MOON and supplying extension points to particular languages.
- *Guarantee the behavior preservation.* From an approach based on contracts (pre, postconditions and invariants) and checking that the system follows generating the same observable results, once carried out the refactoring (using testings, in the line of black box testing [8].)
- *Refactoring application.* Manual, semi-automatic (partially assisted by tools) or automatic, from the point of view of the programmer.

[2] MOON is an acronym of Minimal Object-Oriented Notation.

- *Consequences for client classes.* Pointing out the effects of the refactoring execution for the classes.
- *Consequences for objects.* Showing the effects of the refactoring on the previous persistent objects.

Moreover, in previous works [21], we observed an absence of works and catalogs on generic refactorings, with the exception of *parameterize* refactoring [2]. The present incorporation of the concept of parametric types and generic classes in widely extended languages as JAVA and C#, is a very interesting field to extend the works initiated in [14, 21], developing a subcatalog of refactorings related to generic aspects.

4 Goals

From the information available in the model language MOON, we propose a research of a refactoring catalog from the taxonomies proposed by [3, 22].

This analysis must obtain:

- *A formal support to the definition of those refactoring that can be achieved with language independence, preserving the behavior.* Identifying *"when"* and *"how"* to accomplish the refactoring with OOL independence, as well as those language dependent factors which block the refactoring execution:
  - Support to obtain metric measures.
  - Metric selection associated to bad smells.
  - Refactoring inference based on metric values and bad smells.
  - Analysis of the refactoring elements.
  - Refactoring definition with certain language independence.
- *Definition of the refactoring elements, taking MOON as support.* The aim is to implement these detected elements integrated in the refactoring definition:
  - Support to refactoring implementation.
  - Reuse of the refactoring elements.
- *Measure the software quality improvement.* We must justify refactoring from metric observations:
  - Improvement or change of metric values after refactoring.

A new approach detecting refactoring opportunities is reached using language independent metrics and other techniques. Recommended metric values will be fixed by programmer or application domain.

By other side, once this aims have been reached, we have a solid base to support a tool that allows to reuse the effort achieved in the independent language definition refactoring. This can be used in IDEs with multilanguage support, and also in refactoring tools applied to different languages. Analyzed refactorings could be executed in a set of object oriented language that could be projected on MOON. It is not necessary to repeat the whole process again.

5 Approach

To carry forward this work, it is necessary to select those refactoring from Fowler's catalog [8] included currently in the most widely used IDEs, and by other side, work in a new refactoring catalog related with generic aspects on object-oriented languages.

From the refactoring analysis, the result of this work gives a solid base to develop a prototype that includes the next features:

1. Identify *when* achieve the analyzed refactorings.
2. Assist to the refactoring in an automatic mode, or partially assisted by the programmer, preserving the behavior.
3. Implement the analyzed refactoring, using an approach with and for reuse from the elements defined in refactoring. Also we must supply some mechanisms to undo the refactoring effect.
4. Set a software quality metric to measure the improvement after refactor.

To establish the results obtained with this prototype, we pretend to compare the common refactorings in the current IDEs that work in statically typed object-oriented languages more extended (`Java`, `C#`, `Eiffel`), analyzing in which points we have improved the results, following the taxonomies in [3, 22]. Also we must take into account the number of languages that supports this proposal.

The final result must be a prototype that include refactoring on a wide set of codes, as well as a saving of time in the development of tools with the same set of refactoring on similar languages.

6 Current Progress

We accomplish the refactoring definition on the model language according to a template [8, 23]. The template is composed by a name, a brief description, motivation, inputs, preconditions, actions and postconditions.

The preconditions and postconditions are defined on the basis of a set of logical predicates and functions. It is necessary that all the definition of preconditions, functions, actions and postconditions can be expressed on the basis of the information represented in the model language MOON.

Some of them depend on a concrete language peculiarity, so they are analyzed and extended, if it is possible. In other case, they are classified as "not possible to be defined".

Once this process is finished on a subset of refactoring [21], we can discover which refactoring operations can be independently performed of the source language.

The prototype that supports this refactoring definition needs parsers from a concrete language to MOON. For now, we have a basic parser from Java 1.4 to MOON, and the prototype is able to achieve some simple refactorings, including some refactoring based on generic features (using GJ [24]).

Our work follows this line, developing more parsers (Java 1.5 and C# 2.0 could be the obvious candidates) and reusing the previous refactoring definition. Our next step must be also introduce metrics on MOON, referencing to "Bad Smells" [8], to evaluate *when*, *how* and *why* we must refactor with language independence.

References

1. Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings ISPSE 2000*, pages 157–167. IEEE, 2000.
2. Yania Crespo. *Incremento del potencial de reutilización del software mediante refactorizaciones.* PhD thesis, Universidad de Valladolid, 2000. Available at http://giro.infor.uva.es/docpub/crespo-phd.ps.
3. Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.
4. Eduardo Casais. Managing class evolution in object-oriented systems. Technical report, Centre Universitaire dInformatique, University of Geneve, 1990.
5. Eduardo Casais. *An Incremental Class Reorganization Approach*, volume LCNS 615, pages 114–132. Springer-Verlag, 1992.
6. Eduardo Casais. *Automatic reorganization of object-oriented hierarchies: a case study*, pages 95–115. Springer-Verlag, 1994.
7. William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1992.
8. Martin Fowler. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 2000.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
10. Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, IL, USA, 1999.
11. Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* PhD thesis, University of Bern, 2001.
12. Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. MOOSE: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, 2000.
13. Carlos López, Raúl Marticorena, and Yania Crespo. Hacia una solución basada en frameworks para la definición de refactorizaciones con independencia del lenguaje. In *Actas JISBD'03, Alicante, España*, November 2003.
14. Raúl Marticorena, Carlos López, and Yania Crespo. Refactorizaciones de especialización en cuanto a genericidad. Definición para una familia de lenguajes y soporte basado en frameworks. In *Actas PROLE'03, Alicante, España*, November 2003.
15. Alex Sellink and Chris Verhoef. Generation of software renovation factories from compilers. In *ICSM*, pages 245–255, 1999.
16. Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, pages 91 – 100, Benvento, Italy, 2003. IEEE Computer Society.
17. Tom Tourwé and Tom Mens. Automated Support for Framework-Based Software Evolution. In *Proc. Int'l Conf. Software Maintenance*, 2003.

18. Pierre Crescenzo and Philippe Lahire. Customisation of inheritance. In *Inheritance Workshop at ECOOP 2002*, pages 23–29. Black et al. Eds, Information Technology Research Institute, Jyvaskyla University Press, 2002.
19. Tom Mens and Michele Lanza. A graph-based metamodel for object-oriented software metrics. In Tom Mens, Andy Schürr, and Gabriele Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.
20. Ralf Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.
21. Raúl Marticorena and Yania Crespo. Refactorizaciones de especialización sobre el lenguaje modelo MOON. Technical Report DI-2003-02, Departamento de Informática. Universidad de Valladolid, septiembre 2003. Available at http://giro.infor.uva.es/docpub/marticorena-tr2003-02.pdf.
22. Yania Crespo and José Manuel Marqués. Definición de un marco de trabajo para el análisis de refactorizaciones de software. *Actas VI Jornadas de Ingeniería del Software y Bases de Datos*, pages 297–310, 2001.
23. Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. *Journal of Automated Software Engineering*, 8:89–120, 2001. This is an enlarged version of ASE Conference paper, October 1999.
24. Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the java programming language: Participant draft specification, 2001.